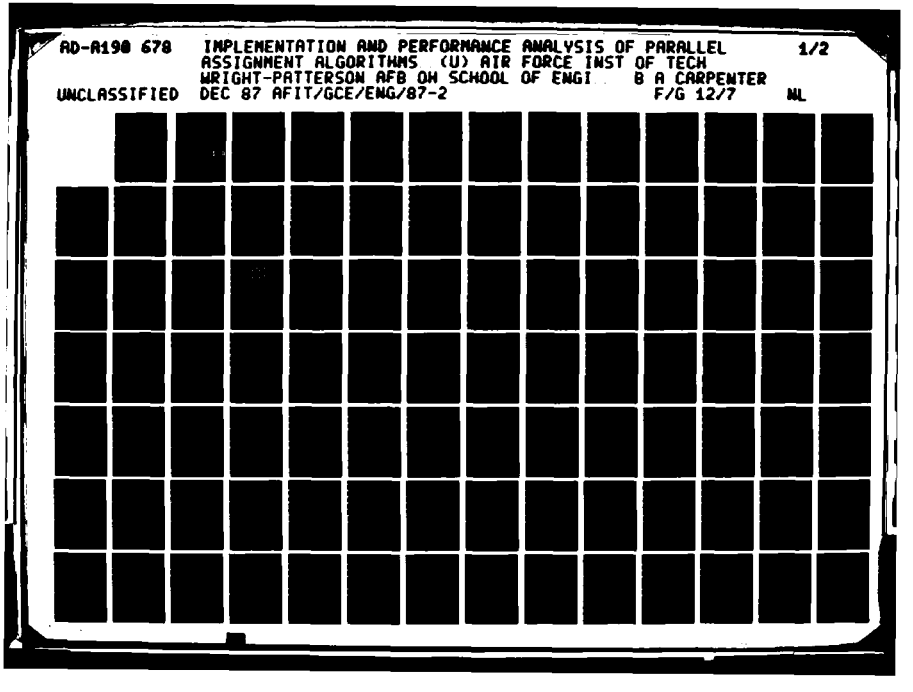
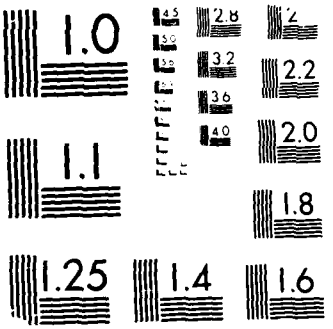

AD-A190 678 IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI B A CARPENTER
1/2
UNCLASSIFIED DEC 87 AFIT/GCE/ENG/87-2 F/G 12/7 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS ON A HYPERCUBE COMPUTER

THESIS

Barry Austin Carpenter
Captain, USAF

AFIT/GCE/ENG/87-2

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

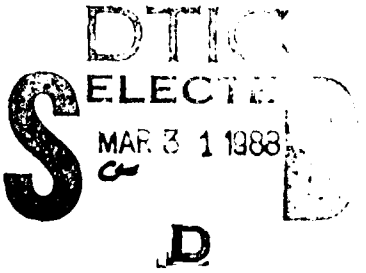
AFIT/GCE/ENG/87-2

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS ON A HYPERCUBE COMPUTER

THESIS

Barry Austin Carpenter
Captain, USAF

AFIT/GCE/ENG/87-2



Approved for public release; distribution unlimited

AFIT/GCE/ENG/87-2

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS ON A HYPERCUBE COMPUTER

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Barry Austin Carpenter, B.S.
Captain, USAF

December, 1987

Acknowledgments

The results of the research presented in this thesis required a great deal of effort. However, I could not have completed this work without some assistance from others. I would first like to thank my thesis advisor, Capt Nathaniel J. Davis IV, who guided me through this research and endured reading and revising many intermediate versions of this thesis. I also wish to thank my committee members, Dr. Tom Hartrum and Dr. Gary Lamont, who were very cooperative and helpful in providing assistance during my research and experimentation. A word of thanks is also due to Mr. Rick Norris and Ms. Juanita Blackford who provided assistance and support in the parallel processing laboratory. Finally, I thank my wife Rhonda who not only assisted me with the typing, but supported me with her understanding and patience during the many long days and nights I spent completing this work.

In addition to the individual assistance, the Strategic Defense Initiative Organization (SDIO) provided the funds to purchase the Intel hypercube computers used to conduct this research and the Sun workstations used to produce this document.

Barry Austin Carpenter

Table of Contents

Page

Acknowledgments ii

Table of Contents iii

List of Figures viii

List of Tables ix

Abstract xii

1. Introduction 1

1.1 SDI And Parallel Computing 2

1.2 The Assignment Problem 4

1.3 Research Objectives 5

1.4 Scope 6

1.5 Assumptions 7

1.6 Overview of the Thesis 8

2. Parallel Processing Background 10

2.1 Parallel Processor Architectures 10

2.1.1 Flynn's Classification of Architectures 11

2.1.2 Types of MIMD Architectures 13

2.1.3 The Hypercube Interconnection Network 13

2.2 The Intel iPSC Hypercube Computer 14

2.2.1 History 15

2.2.2 Hardware Organization 15

Page

2.2.3 Software Development Environment 16

2.3 MIMD Mapping Techniques 17

2.3.1 The Basic Approach 17

2.3.2 Communications Overhead 17

2.3.3 Problem Partitioning 18

2.3.4 Load Balancing 19

2.3.5 Use of Sequential Algorithms 20

2.4 Other Implementations 20

2.4.1 Boost-Phase Track Initiation Algorithms 21

2.4.2 Parallel Branch and Bound 21

2.4.3 The Traveling Salesman Problem 21

2.4.4 Gaussian Elimination 22

2.4.5 Vision Algorithms 22

2.5 Summary 23

3. Development of the Parallel Assignment Algorithms 24

3.1 The Assignment Problem 24

3.1.1 History 25

3.1.2 Statement of the Assignment Problem 25

3.2 Sequential Assignment Algorithms 26

3.2.1 The Simplex Method 26

3.2.2 The Transportation Method 27

3.2.3 The Alternating Basis Algorithm 28

3.2.4 The Hungarian Method 29

3.2.5 Modifications to the Hungarian Method 30

3.2.6 The Branch and Bound Algorithm 31

3.2.7 The Out-of-Kilter Algorithm 32

3.3 Evaluation of Candidate Algorithms 33

Page

3.3.1 The Transportation Method 33

3.3.2 The Hungarian Method 36

3.4 Parallel Combination Strategies 39

3.4.1 Branch and Bound 39

3.4.2 Alpha-Beta Search 40

3.4.3 Divide-and-Conquer 41

3.5 Results of the Analyses 42

3.5.1 Selection of Search Technique 42

3.5.2 Selection of Candidate Algorithm 43

3.5.3 Interprocess Communications Protocol 46

3.5.4 The Parallel Assignment Algorithms 47

3.6 Summary 47

4. Implementation of the Assignment Algorithms 48

4.1 The Experimental Model 49

4.1.1 Assumptions 49

4.1.2 Model Definition 50

4.1.3 The Ballistic Missile Defense Simulation Program 50

4.1.4 Method of Input Data Generation 51

4.2 Sequential Assignment Algorithm Implementations 52

4.2.1 The Sorting Method 52

4.2.2 The Bourgeois and Lassalle Algorithm 54

4.3 Parallel Assignment Algorithm Implementations 56

4.3.1 The First Level: No Communications 56

4.3.2 The Second Level: Partial Communications, Single Iteration 58

4.3.3 The Third Level: Partial Communications, Multiple Iterations 60

Page

4.3.4 The Fourth Level: Parallel Matrix Operations . . 62

4.4 Summary 66

5. Experimental Results and Performance Analysis 68

5.1 Testing Approach 68

5.1.1 Performance Criteria 69

5.1.2 Method of Data Collection 72

5.2 Presentation of Results 73

5.2.1 Level 0 73

5.2.2 Level 1 75

5.2.3 Level 2 79

5.2.4 Level 3 82

5.2.5 Level 4 84

5.3 Analysis of Results 87

5.3.1 Computation Times and Speedup 87

5.3.2 Interprocessor Communications 89

5.3.3 Problem Scalability 91

5.3.4 Cost and Effectiveness of Assignments 94

5.4 Summary 99

6. Conclusions and Recommendations 100

6.1 Parallel Processing: Lessons Learned 101

6.2 Areas of Application 102

6.3 Recommendations for Further Research 103

A. Appendix A: The Transportation Method 105

B. Appendix B: The Hungarian Method 122

C. Appendix C: Additional Results 130

Page

Bibliography 143

Vita 148

List of Figures

Figure		Page
2-1.	Flynn's Classifications (a) SISD (b) MISD (c) SIMD (d) MIMD . . .	12
2-2.	Three-Dimension Cube Structure	15
4-1.	Processor Communication Paths for the First Level	57
4-2.	Communication Paths for the Second and Third Levels	59
4-3.	Communication Paths for the Fourth Level	64
5-1.	Speedups over the Level 0 Implementation (1:1 Weapon-Target Ratio)	88
5-2.	Speedups over the Single-Node Level 1 Implementation (1:1 Ratio) .	90
5-3.	Speedups over the Single-Node Level 1 Implementation (1:10 Ratio) .	91
5-4.	Level 1 Speedups over the Sequential B&L Algorithm (96 Wpns) . .	93
5-5.	Weapon Effectiveness vs. Number of Processors (1:1 Ratio)	95
5-6.	Weapon Effectiveness vs. Number of Processors (1:10 Ratio)	96
5-7.	Assignment Cost vs. Number of Processors (1:1 Ratio)	97
5-8.	Assignment Cost vs. Number of Processors (1:10 Ratio)	98

List of Tables

Table		Page
2-1.	Processor Binary Addresses	14
3-1.	Example Transportation Table	34
5-1.	Timing and Costs of the Level 0 Implementation	74
5-2.	Timing and Speedups of the Level 1 Implementation	76
5-3.	Assignment Results of the Level 1 Implementation	79
5-4.	Timing and Speedups of the Level 2 Implementation	80
5-5.	Assignment Results of the Level 2 Implementation	82
5-6.	Timing and Speedups of the Level 3 Implementation	83
5-7.	Assignment Results of the Level 3 Implementation	85
5-8.	Timing and Speedups of the Level 4 Implementation	86
5-9.	Assignment Results of the Level 4 Implementation	87
A-1.	Initial Transportation Table	108
A-2.	Initial Basic Feasible Solution	109
A-3.	Initial ϵ Assignments	110
A-4.	Initial Row Indicator R_i Assignment	110
A-5.	Calculation of Additional R_i and K_j Values	111
A-6.	Additional R_i and K_j Values	111
A-7.	Complete R_i and K_j Values	112
A-8.	Δ_{ij} Values for Unassigned Cells	112
A-9.	θ-Path for Exchange of Variables	113
A-10.	New Assignment from First Iteration	114
A-11.	Second Nondegenerate Basic Solution	114
A-12.	Second Set of R_i and K_j Variables	115
A-13.	Second Set of Δ_{ij} Values	115

Table Page

A-14.Second θ-Path for the Most Negative Δ_{ij} 116

A-15.Second Iteration Assignment 117

A-16.Third Nondegenerate Basic Feasible Solution 117

A-17.Third Set of R_i and K_j Variables 118

A-18.Third Set of Δ_{ij} Variables 118

A-19.Third θ-Path for the Most Negative Δ_{ij} 119

A-20.Third Iteration Assignment 119

A-21.Third Iteration R_i, K_j, and Δ_{ij} Values 120

A-22.Results of Transportation Method 120

B-1. Results of Subtracting Minimum Row Elements 124

B-2. Results of Subtracting Minimum Column Elements 125

B-3. Independent Null Row Elements 125

B-4. Independent Null Row and Column Elements 126

B-5. Checked Rows Without Boxed Null Elements 126

B-6. Checked Columns with Null Elements in Checked Rows 127

B-7. Checked Rows with Boxed Null Elements in Checked Columns 127

B-8. A Minimum Set of Covering Lines 127

B-9. New Table From Step 5 128

B-10.Independent Null Row and Column Elements 128

B-11.Results of Hungarian Method 129

C-1. Timing and Speedups of the Level 1 Implementation (32 Wpns) . . . 130

C-2. Timing and Speedups of the Level 1 Implementation (64 Wpns) . . . 131

C-3. Timing and Speedups of the Level 1 Implementation (128 Wpns) . . 132

C-4. Timing and Speedups of the Level 2 Implementation (32 Wpns) . . . 132

C-5. Timing and Speedups of the Level 2 Implementation (64 Wpns) . . . 133

C-6. Timing and Speedups of the Level 2 Implementation (128 Wpns) . . 133

C-7. Timing and Speedups of the Level 3 Implementation (32 Wpns) . . . 134

Table Page

C-8. Timing and Speedups of the Level 3 Implementation (64 Wpns) . . . 134

C-9. Timing and Speedups of the Level 3 Implementation (128 Wpns) . . 135

C-10.Timing and Speedups of the Level 4 Implementation (32 Wpns) . . . 135

C-11.Timing and Speedups of the Level 4 Implementation (64 Wpns) . . . 136

C-12.Timing and Speedups of the Level 4 Implementation (128 Wpns) . . 136

C-13.Assignment Results of the Level 1 Implementation (32 Wpns) 137

C-14.Assignment Results of the Level 1 Implementation (64 Wpns) 137

C-15.Assignment Results of the Level 1 Implementation 128 Wpns) 138

C-16.Assignment Results of the Level 2 Implementation (32 Wpns) 138

C-17.Assignment Results of the Level 2 Implementation (64 Wpns) 139

C-18.Assignment Results of the Level 2 Implementation (128 Wpns) . . . 139

C-19.Assignment Results of the Level 3 Implementation (32 Wpns) 140

C-20.Assignment Results of the Level 3 Implementation (64 Wpns) 140

C-21.Assignment Results of the Level 3 Implementation (128 Wpns) . . . 141

C-22.Assignment Results of the Level 4 Implementation (32 Wpns) 141

C-23.Assignment Results of the Level 4 Implementation (64 Wpns) 142

C-24.Assignment Results of the Level 4 Implementation (128 Wpns) . . . 142

Abstract

The process of effectively coordinating and controlling resources during a military engagement is known as *battle management/command, control, and communications* (BM/C3). One key task of BM/C3 is allocating weapons to destroy targets. The focus of this research is on developing parallel methods to achieve fast and cost effective assignment of weapons to targets. Using the sequential Hungarian method for solving the assignment problem as a basis, this report presents the development and relative performance comparison of four parallel assignment algorithms implemented on the Intel iPSC hypercube computer.

The first approach partitions the problem space into smaller, independent subproblems and assigns each to a processing node in the hypercube. The second and third approaches also partition the problem space, but they assign each partition to a group of processing nodes. Each group is controlled by a separate node which further subdivides the partition among members of the group. In the second approach, the control node acts as an arbitrator to eliminate the redundant assignment of weapons to targets by idling redundantly allocated weapons. The third approach eliminates redundant weapon allocations by selecting the least costly redundant allocations and directing additional processing to reallocate the more costly weapons. The fourth approach is a parallel implementation of the Hungarian algorithm, where certain subtasks are performed in parallel. This approach produces an optimal assignment instead of the sub-optimal assignment generally obtained using either of the three heuristic approaches.

The relative performance of the four approaches is compared by varying the number of weapons and targets, the number of processors used, and the size of the problem partitions. The first and second approaches produce assignment solutions significantly faster than the baseline sequential methods. The third and fourth ap-

proaches yield slower solutions, but are faster than sequential methods of assignment.

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS ON A HYPERCUBE COMPUTER

1. Introduction

Parallel processing is a method of computation that exploits the concurrent events that occur in the solution of many different problems [HwB84]. Parallel computers employing multiple processors exploit these concurrent events by assigning each event to a different processor for simultaneous processing. The results of these parallel computations are combined to form a solution to the overall problem [Hil87]. Parallel processing is presently the subject of intense research and development. The main reason for the increased interest in parallel processing is the wider availability of parallel multiprocessor computers [Fre86]. Improved technology in the areas of VLSI (Very Large Scale Integration) circuits, high speed communications, and hardware packaging have combined to make these parallel computers more reliable and much less expensive [Sei85, Den86, Fre86].

Recent software implementations have shown that significant reductions in processing times are possible using parallel processing [Qui87]. Many of these implementations involve large scale problems in areas such as fluid dynamics [EbB86], high energy physics [Fox84], partial differential equation solutions [SaN85], statistical mechanics [FoO84], image processing [MuA87], and several other areas that were previously not feasible because of the excessive processing times required when using single-processor computers. Faster solutions to these large scale problems appeal to many researchers in government and industry because they allow more accurate and extensive modeling of complex processes during the development and design phases

of new systems. One particular government organization with a keen interest in the increased processing speeds provided by parallel processing is the Strategic Defense Initiative Organization (SDIO) [AdW85, Lin85, BoR85].

1.1 SDI And Parallel Computing

The Strategic Defense Initiative (SDI) was launched by President Reagan in a televised speech on March 23, 1983. In this speech, he challenged scientists and engineers to work to render nuclear weapons "impotent and obsolete." He proposed a research and development program to determine if a "smart" system of nonnuclear defense could effectively knock out incoming offensive ballistic missiles before they detonate over our country [Rea83]. If all dollar amounts are adjusted to today's value, the SDI is potentially the most expensive research and development program ever attempted and far more expensive than the Manhatten Project which produced the atomic bomb [AdF85].

The overall system architecture of the SDI system is envisioned as one of several defensive layers corresponding to the different phases that occur in the trajectory of a ballistic missile. Those phases are the boost phase, the midcourse phase, and the reentry or terminal phase [DrF85]. Within each defensive layer, computers will use information gathered from sensors to detect, classify, and track potential targets. Using this information and predefined engagement strategies, weapons will be assigned to destroy certain high-threat targets. After firing on assigned targets, the effectiveness of the weapons would be evaluated and used to make future weapon engagement decisions. The combination of all of these processes is known as battle management/command, control and communication or BM/C3 [SeD85, Lin85].

Many prominent scientists argue that the realization of a reliable defense system of the magnitude that will be required by the SDI is not possible [Lin85, Par85, Noz86]. Although development in the areas of laser beam, particle beam, and kinetic energy weapons is still in the beginning stages, preliminary results are promising.

A major issue with these weapons is providing them sufficient energy for effective operation when they are deployed in space [AdF85]. Development of the BM/C3 system is the area of most concern. During a full-scale missile attack, hundreds of thousands of interrelated decisions will need to be made about how to most effectively utilize available defensive weapons. These complex decisions must be made within milliseconds of each other in order to deploy defensive weapons in a timely manner. Because time and complexity constraints make them humanly impossible, these decisions must be made with the assistance of fast and reliable computers using intelligent software. For example, if the enemy launched 1400 missiles in an attack, then more than 10 enemy missile kills per second would be needed to destroy most of the missiles shortly after they were launched [AdW85]. Development of the millions of lines of error-free software code and the computer systems to flawlessly execute the software to accomplish these BM/C3 tasks is viewed as impossible by Parnas [Par85]. The magnitude and complexity of BM/C3 software prompted Lieutenant General James A. Abrahamson, director of the SDIO, to state in an interview that the "incredible software problem" of the battle management system is "the challenge of all time" [Chr85]. The SDIC is now actively conducting research in many areas on how to meet the challenge of developing a viable battle management system.

An area of particular interest is the development of fast and reliable BM/C3 computer systems for controlling weapons, sensors, and other equipment that will comprise the SDI system [AdW85]. One concern is the computation time that a single-processor computer might require to control and coordinate all of the activities within a defensive layer. The basic computational speed of a single processor is limited by internal signal propagation delays and is not expected to exceed 1 GFLOPS (Giga-Floating Point Operations Per Second) with current circuit technology [Den86]. Estimates of the computational speed required for some BM/C3 tasks are more than 10 GFLOPS [AdW85]. One way researchers believe faster computations will be possible is to develop system architectures that utilize parallel-

processors [SeD85]. The defensive layer could then be divided into relatively independent regions. Each region would be assigned to a separate set of processors within the multiprocessor computer to coordinate and evaluate activities within that region. When combined with efficient software developed especially for parallel-processors, the overall computations could be completed in a time much shorter than that achievable with a single-processor computer [San87]. The ideal speed increase or speedup of a parallel-processor with n processors over a single-processor computer is n. In some cases, greater than n speedup can be achieved by utilizing certain parallel algorithms. The possibility of ideal or better speedups with parallel computers creates the potential for meeting or exceeding the predicted computational requirements of the proposed BM/C3 system.

1.2 The Assignment Problem

One of the critical BM/C3 tasks is the assignment of weapons to targets. Situations similar to the problem of assigning weapons to targets frequently occur in other areas such as operations research, logistics management, and even in a computer's internal management of its resources. Typically, there exists a number of resources available to be allocated to a number of requesters. In most cases, there are more requesters than there are resources. In cases such as these, decisions must be made as to which requesters are allocated resources and which requesters are denied resources. The problem is generally known in the literature as the assignment problem and usually involves allocating available resources to competing requesters in such a way as to maximize some measure of profit or award, or to minimize some measure of penalty [Kuh55, Chu57, Kur62].

The assignment problem can be solved in many different ways. The brute force method would be to enumerate all the possible ways resources could be allocated to requesters and then choose the combination that provides the best allocation. This method might work well for a very small number of resources and requesters, but for

any realistically sized system, the time required to enumerate all of the possibilities would be prohibitive. For example, if there were only 20 resources and 20 requesters, the number of different resource-to-requester assignments would be 20! or 2.433×10^{18} [Chu57]. This difficult problem has been recognized by mathematicians and computer scientists who have developed algorithms that provide more time efficient methods of arriving at the best, or very close to the best, allocation of resources.

Research on developing algorithms to solve the assignment problem has a long history. Von Neumann, who is considered the inventor of the conventional single-processor computers used today, experimented with the computational advantages of using linear programming techniques to solve the assignment problem [Kuh55]. Several others have also conducted research, developed algorithms, and devised software implementations to achieve faster and more efficient methods of solving the assignment problem [Mun57, Kur62, LaM69, SrT72, SrT73, Hat75, Hun83, McG83, MaN86]. The techniques involved with many of these research efforts are similar and involve linear programming, graph theory, and set theory.

1.3 Research Objectives

Although algorithms have been developed to solve the assignment problem, all of them have been implemented as sequential processes. Because these algorithms are sequential in nature, they are easily implemented on sequential, single-processor computers. Unfortunately, algorithms that solve the assignment problem in a parallel processing environment have not yet been developed. Given the potential speedups possible with parallel computers, it would seem advantageous for the battle management portion of the SDI system to use a parallelized version of one of these sequential algorithms to perform the weapon-target assignment task. This research first investigates the techniques for mapping algorithms onto parallel-processors. Then, sequential assignment algorithms are analyzed to select a candidate for parallelization.

The primary objective of this thesis investigation is implementation of assignment algorithms on a parallel multiprocessor computer. After successful implementation, the performance of the parallel algorithms is analyzed. In this analysis, particular attention is focused on the effects of inter-processor communications, load balancing among processors, execution times, and machine size to problem size relationships. The parallel computer used for the implementations is the Intel iPSC (Intel Personal Super Computer) multiprocessor system which is described in detail in Chapter 2.

1.4 Scope

In this study, the problem of assigning weapons to targets in a parallel processing environment is the primary focus. For this reason, exact details of the battle management system such as how the individual targets are detected and tracked; the specifics of particular weapons; the operation and sensitivity of sensor devices; and the three-dimensional and rotational characteristics of weapon-to-target geometry are not addressed. These factors are accounted for to a certain degree by using techniques described in Section 1.6 (Assumptions). However, the concepts that are explored in this study should contribute to the research and development of future battle management systems. The specific steps of this research are as follows:

1. First, techniques for partitioning and mapping sequential algorithms onto parallel computer architectures are researched. From the candidate techniques, one is chosen that best matches the loosely coupled architecture of the Intel hypercube.

2. The study continues by locating efficient sequential algorithms that solve the assignment problem. These algorithms are evaluated to determine which ones, if any, lend themselves to parallel implementation.

3. Using the chosen algorithm and mapping technique, weapon-target assignment programs are designed and then implemented using the parallel "C" program-

ming language supported by the Intel iPSC. A top-down, structured approach to software development is used to minimize the time required for implementation.

4. The implementations are tested on the Intel hypercube machine using different numbers of processors and varying processor configurations. The performance is measured with varying numbers of weapons and potential targets to generate ample data for analysis and comparison of the different implementations.

1.5 Assumptions

A number of simplifying assumptions were necessary in order to both limit the detail of the research to a reasonable level and still allow time for completion. First, the number and location of potential targets, along with their relative importance, are assumed to be available on demand. Likewise, the number and status of available resources or weapons are also assumed to be immediately available when requested. Problems associated with detecting and classifying potential targets, and the details of evaluating the effectiveness of weapons already assigned to targets are not considered, although simulated results of those functions are supplied as input data to the programs. Weapons are considered to be reuseable with a finite number of "shots," and are assignable to one target at a time for a single "shot." Each instance of assignment is assumed to be one "snapshot" of the dynamic process of missiles in some phase of their trajectory.

Because the main focus of this study is on the implementation of a parallel weapon-target assignment algorithm, an entirely realistic simulation of missile trajectories and distribution patterns of missiles within the different regions is not attempted. However, plausible missile attack scenarios are generated by an unclassified ballistic missile defense simulation program. These scenarios are used as a basis for constructing similar data as input to the implementations developed in this study. Factors such as space-based weapons platform orbits, rotation of the earth, and plausible missile trajectories originating from locations in the Soviet Union are accounted for in the simulation program [Odo85].

1.6 Overview of the Thesis

This chapter completes a brief overview of the SDI, parallel computing, and general assignment problems. The objectives of this research were presented along with the scope, assumptions, and the general approach to be taken to reach the stated objectives. The remainder of this thesis develops in detail the steps listed in Section 1.5. Chapter 2 begins with a brief survey of the different types of parallel computers and then uses the survey as a basis for describing the Intel iPSC parallel computer. It continues with an investigation of the techniques for developing parallel software implementations for the Intel iPSC and concludes with a summary of the techniques selected for use. In Chapter 3, a thorough presentation of assignment algorithms developed in the past three decades is presented. Then, development of the parallel assignment algorithm begins by using the techniques selected in Chapter 2 and any useable portion of the assignment algorithms developed by others in the past. Chapter 4 begins with a detailed definition of the experimental model, a presentation of the ballistic missile defense (BMD) simulation program and a description of a method for generating target scenario data using the BMD simulator as a basis. Chapter 4 continues with a description of the different implementations of the parallel assignment algorithm. In Chapter 5, the method of testing and data

acquisition is explained first, followed by a presentation of the results obtained from performance runs on the Intel iPSC. After presenting the results, detailed analyses of these results are performed. Chapter 6 ends the thesis with conclusions and recommendations for further study.

2. *Parallel Processing Background*

Chapter 1 briefly introduced the subject of parallel processing. This chapter continues with a more in-depth discussion of parallel processing by first surveying the different types of parallel-processor architectures and then focusing on a particular class of architecture known as Multiple Instruction-stream, Multiple Data-stream (MIMD). Then the history of development, the hardware, and some of the important features of the Intel iPSC hypercube computer are all presented. Techniques for mapping problem solutions onto parallel processor architectures are then investigated, followed by a discussion of the problems associated with parallel algorithm implementations. This chapter concludes with a presentation of recent implementations by others on MIMD parallel-processor computers.

2.1 Parallel Processor Architectures

The Von Neumann machine is a sequential computer consisting of a central processing unit (CPU), a memory system, and an input/output (I/O) system. Instructions are accessed from the memory system and executed in the CPU one at a time. This Von Neumann model of a sequential computer is the underlying architecture of a majority of the conventional computers available today [EbB86]. Steady improvements in VLSI technology have allowed this sequential architecture to remain popular by reducing the signal propagation delays, discussed in Chapter 1, between the CPU and memory [EbB86]. However, reducing signal propagation delays is becoming increasingly more difficult because the physical limits of signal transmission speed in silicon, the most common fabrication material, is being approached [Den86]. This is one motivation behind the development parallel computer architectures to achieve faster processing times instead of attempting to speed up sequential Von Neumann computers.

A processing element (PE) can be basically defined as a CPU and a local memory unit for storing programs and local data. Parallel computer architectures utilize a number of processing elements, usually in the form of Von Neumann machines, that are linked together by an interconnection network. This interconnection network provides a means to either transfer information between the different processing elements or to allow access to a common data storage area. The following sections discuss the different types of parallel architectures.

2.1.1 Flynn's Classification of Architectures Flynn classified computer architectures into four categories according to the number of instruction and data streams utilized [Fly66]. Those categories, are Single Instruction-stream Single Data-stream (SISD), Single Instruction-stream Multiple Data-stream (SIMD), Multiple Instruction-stream Single Data-stream (MISD), and Multiple Instruction-stream Multiple Data-stream (MIMD). The SISD category describes the sequential Von-Neumann machines. MISD is generally regarded as an impractical classification of a computer architecture [HwB84]. The SIMD and MIMD categories describe the architectures of parallel computers. Representations of these classifications are shown in Figure 2-1.

SIMD machines are generally comprised of a number of simple processing elements statically linked to a central control unit that interprets instructions and issues commands to the processing elements. Processing in parallel SIMD machines is usually characterized by identical operations simultaneously performed in lock step on each element of an array or matrix. The Illiac IV, one of the first SIMD machines developed in the 1960's, was used to solve problems in areas such as fluid flow, aerodynamics, and meteorology [RiS84]. A recently introduced SIMD computer is the Connection Machine, which employs 65,536 simple processors [Hil87].

In contrast to SIMD machines, the individual processors in MIMD machines do not necessarily perform the same instructions at the same time. Processing elements are relatively independent and each one may be executing a completely different program. Different types of MIMD architectures will be discussed in the next section.

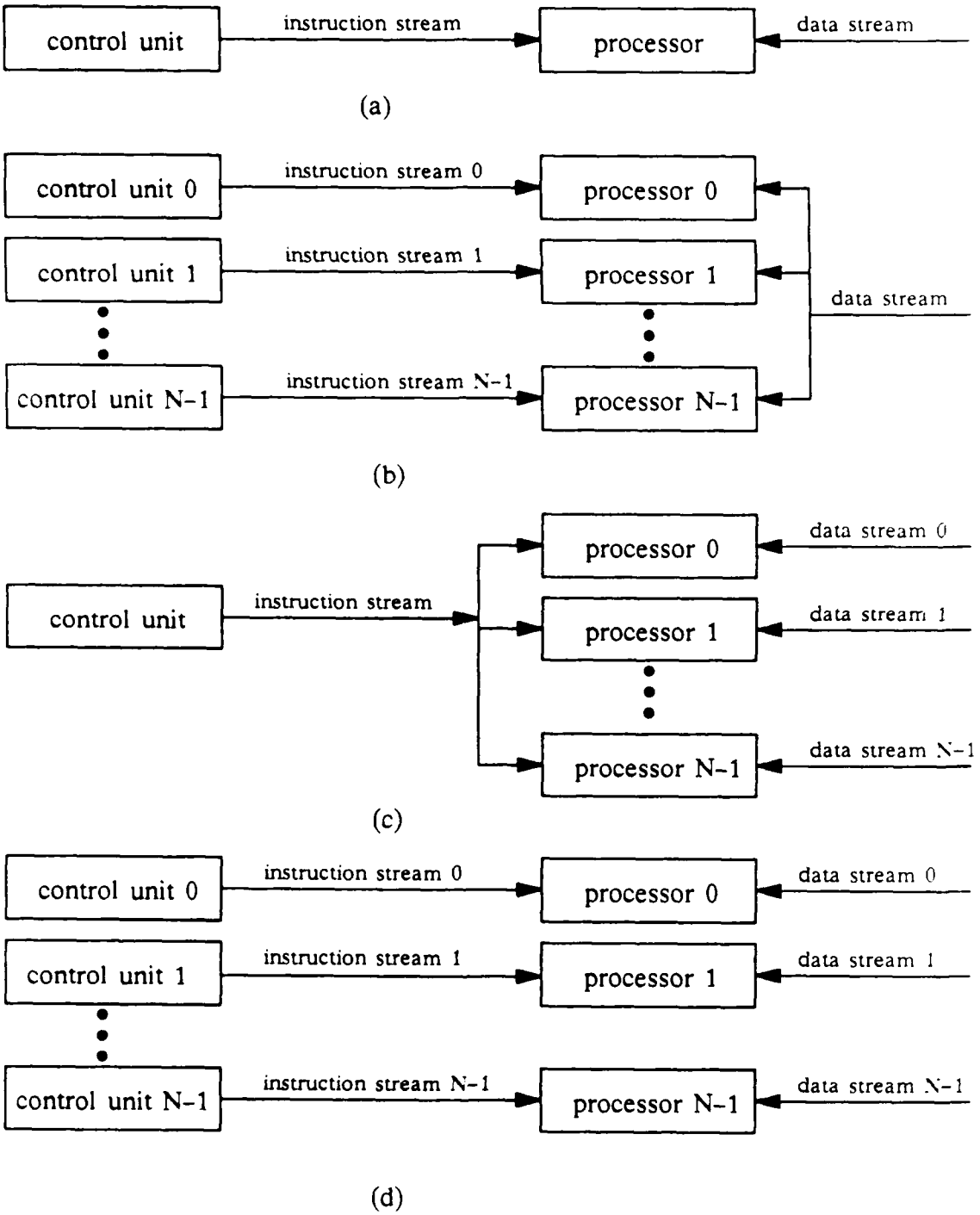


Figure 2-1: Flynn's Classifications (a) SISD (b) MISD (c) SIMD (d) MIMD

2.1.2 Types of MIMD Architectures The MIMD classification of a computer architecture can be further divided into two sub-classes, based on the memory structure and the type of interprocessor communications. One sub-class is the shared memory machine where all individual processing elements have access to a large global memory which is used to access common data and to pass information between processors. Shared-memory machines are also known as tightly coupled processors because of the degree of interaction between processors imposed by the global memory [HwB84].

Another sub-class of MIMD computers is the local memory or loosely-coupled machine. Processors in loosely-coupled machines each possess their own private memory that is not accessible by the other processors. Information is exchanged between processors by passing messages through the interconnection network. Processors in these machines are generally more independent than those in the shared memory machines. Loosely-coupled machines derive their name from the reduced interaction between the individual processors [MuA87]. Many of the commercial MIMD computers available today are loosely coupled [HwB84].

2.1.3 The Hypercube Interconnection Network Parallel solutions to certain problems sometimes require the processors to be configured into a ring, mesh, star, or tree structure [Fen81]. There are a number of ways to interconnect the processors in an MIMD multiprocessor computer. One class of interconnection network that can function as any of the listed configurations is based on the cube interconnection function [Sei85]. The *m cube* function can be defined as:

$$cube_i(p_{m-1}, \ldots, p_1, p_0) = p_{m-1} \cdots p_{i+1}\ \overline{p}_i\ p_{i-1} \cdots p_1\ p_0 \qquad (2-1)$$

where $0 \leq i < m$ and $\overline{p}_i$ denotes the complement of p_i.

The cube function is the basis for networks such as the multistage cube network

[McS85], the Boolean n-cube [Pea77], and the hypercube [SaS85]. The hypercube interconnection scheme has the advantage that if the total number of processors is N, the maximum number of intermediate links that must be traversed by a message from one processor in order to communicate with any other processor in the network is $log_2 N$.

The processors in a hypercube interconnection network are linked together based on the binary representation of the processor's address. Processors whose binary addresses differ by only one bit (i.e., the cube function $cube_i$ for bit i) are connected. For example, in a three-dimensional cube there are $8 = 2^3$ processors. These binary addresses can be represented as shown in Table 2-1.

Table 2-1. Processor Binary Addresses

Processor	*Address*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Each processor is connected to three other processors using this scheme. The resulting structure can be represented by the diagram illustrated in Figure 2-2 where the labeled nodes represent processors and the lines represent the links between the processors. Different dimension hypercubes can be formed by following the same addressing scheme.

2.2 The Intel iPSC Hypercube Computer

The Intel iPSC hypercube computer is used for implementing the parallel as-

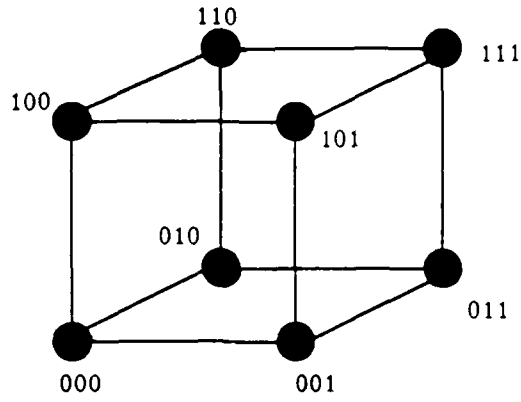


Figure 2-2. Three-Dimension Cube Structure

signment algorithms developed in Chapters 3 and 4 of this study. An overview of the Intel machine and some of its important features is necessary in order to understand some of the decisions that are made during development of the implementations.

2.2.1 History The origin of the Intel iPSC can be traced back to research performed at Caltech and the NASA Jet Propulsion Laboratory during 1978-1981 [Sei85]. This research formed the basis for an MIMD, local memory, multiprocessor machine that was designed and built primarily as a hardware simulation of a computer researchers expect to be able to implement entirely in VLSI in the future. However, the excellent performance of the prototype prompted Seitz and his colleagues to experiment with solving a variety of computationally-intensive problems. They nick-named this new machine the Cosmic Cube [Sei85]. The Cosmic Cube was later developed into a commercial computer system named the iPSC (Intel Personal Super Computer) by the Intel Corporation. Customer shipments of the iPSC began in February 1985 [Den86]. The iPSC is now available to researchers at many centers, including the Air Force Institute of Technology (AFIT).

2.2.2 Hardware Organization The Intel iPSC is available in several configurations ranging from a 16-processor, 4-dimension cube up to a 128-processor, 7-

dimension cube. In the basic configuration, each processor is built around an Intel 80286 microprocessor, an 80287 numeric coprocessor, and 512K of random access memory (RAM) [Int86]. Options such as additional memory and vector-processing capabilities can make the iPSC a very powerful machine for a modest cost when compared to large supercomputers such as the well known and expensive Cray series.

The individual processing elements or nodes are interconnected in a hypercube topology, with communications coprocessors handling the processor-to-processor message passing duties. The user develops applications for and communicates with processors in the cube through an intermediate host known as the cube manager. The cube manager is also built around the 80286 microprocessor and 80287 coprocessor, but has additional memory capacity [Int86].

2.2.3 Software Development Environment The software development environment of the iPSC is based on a derivative of the UNIX operating system known as the XENIX environment [Int86] The languages supported are parallel versions of FORTRAN, C, and Lisp. Applications are developed using the cube manager as a means to compile, debug, and run programs written in these modified languages. Predefined library functions are used to perform operations such as opening communications channels between the cube manager or other processors, sending or receiving either synchronous or asynchronous messages from other processors, controlling processes running on processors in the cube, and many other functions unique to the Intel hypercube. A program to simulate the functions of the iPSC hypercube for use in initial program development is available for other systems running the BSD 4.2 UNIX operating system. However, accurate performance data for applications must be obtained using the actual Intel iPSC machine.

2.3 MIMD Mapping Techniques

Much has been written about techniques for developing applications software for parallel MIMD computers. These techniques are sometimes known as mapping techniques [Sei85, Fox84]. The most important mapping techniques and some areas to be concerned with while developing implementations are covered in this section.

2.3.1 The Basic Approach Many science and engineering problems are naturally divided into concurrent processes [Sei85]. If they are relatively independent, either one or several of these processes can be assigned to separate nodes or processors in a parallel computer for concurrent processing. Then the "intriguing and ... amusing" task of coordinating the computing activities in each processor must be devised [Sei85]. Continuing, Seitz says from experience that application formulation for the multiprocessor Cosmic Cube "has not proved to be very much more difficult than it is on sequential [single processor] machines." In many cases, he says parallel applications are based on adaptations of well known sequential algorithms.

Fox and Otto maintain that "the main stumbling block to the use of concurrent processors is the difficulty of formulating algorithms and programs for them." They go on to say "that concurrent processors are quite easy to use and ... address the vast majority of computationally intensive problems." They agree with Seitz when they say that most computationally demanding problems are not solved by using complex algorithms, but "rather there is a relatively simple procedure ... that one must apply to a basic unit ... in a world that consists of a huge number of such units."

2.3.2 Communications Overhead One of the most common problems associated with applications for parallel processing is the minimization of the communications overhead. The ratio of communications to computations should be approximately one (unity) [Fox84]. This means that the amount of communications should not be greater than the computations. An example of communications over-

head is when information about a problem subdomain contained in one processor is needed by another processor working on a different subdomain. Exchange of this information requires that these processors communicate with each other using the interconnection network. This type of communication between processors must be kept as low as possible [Fox84]. "An important measure of an algorithm's efficiency ... [is] ... the time to move the data" [HoZ83]. This "time to move the data" referred to by Horowitz and Zorat is the communications overhead.

Saltz says there are several techniques that can be used to reduce the communications overhead. One technique is to reduce the quantity of information to be communicated by only sending information that is absolutely necessary. Another method is to reduce the frequency of communications by sending several bits of information in each message [SaN85]. Saltz mentions one other method that involves overlapping communications with processing, which can be accomplished by using asynchronous message-passing library functions in the Intel iPSC programming environment. Another technique for reducing communications overhead, related to problem partitioning, involves increasing the size of the subdomain assigned to each processor. This absorbs some of the communications that would have been necessary, but also reduces the level of parallelism [SaN85].

2.3.3 Problem Partitioning According to Fox and Otto, the first step in formulating a solution to large problems on a concurrent processor is to partition the problem into many parts and assign a different part of the problem to each individual node or processor. Part of the difficulty with partitioning the large problem is deciding on the size of the subproblems. If the subproblems are too small, there is a chance that excessive communications between processors will be necessary to complete the solution [FoO84]. On the other hand, they say forming larger subproblems tends to reduce the communications overhead and increase the efficiency of the computations.

Cvetanovic has written a paper discussing the effects of problem partitioning and granularity on multiprocessor performance [Cve87]. She says that as the size of computations performed on the separate processors decreases, the amount of parallel computations increases. But because of the increased parallelism, computations are performed faster and more requests for additional data or communications with other processors are initiated. As the communications increase, the overall processing slows down. According to Cvetanovic, the following parameters are likely to have the most significant effects on multiprocessor performance:

- The amount of parallelism inherent in the application of the problem.
- The method for decomposing a problem into smaller subproblems.
- The method applied to allocate these subproblems to processors.
- The grain size of a subproblem executed on each processor.

Cvetanovic concludes that problem partitioning has a strong effect on multiprocessor performance. If the subproblem size introduces unacceptable communications overhead, she suggests two methods for reducing this overhead. The first method is to increase the capabilities of the interprocessor communications network. This is seldom possible, so the second method she suggests is more promising. It involves increasing the subproblem size in order to transform some interprocessor communications into intraprocessor communications. This transformation effectively reduces the demands on the communications network and increases overall performance.

2.3.4 Load Balancing Another factor to consider in partitioning a problem is the "load balancing" [FeK85, FoO84]. Efficiency is increased if all processors are performing essentially the same computations. The general idea is that the amount of communications between processors is not as important as the "amount of computation done per communication" [FoO84]. Fox says that memory requirements per processor must be equal and fixed in order to ensure the efficiency of the implementation will not depend on the number of processors in the machine. This restriction

achieves load balancing of the processors by insuring that no one processor will perform the bulk of the computations [Fox84].

2.3.5 Use of Sequential Algorithms On the use of sequential algorithms in parallel implementations, Fox says that each processor performs essentially the same computations a single processor computer would perform. The difference is that the computations are performed on a subdomain of the overall problem. He says the development of programs to run on the individual processors of a multiprocessor computer should be very similar to those used in a uniprocessor machine. An exception to this similarity occurs when "boundary conditions" must be considered where the problem domains of programs running in different processors overlap. In cases such as these, interprocessor communications and some type of synchronization must occur in order to complete the solution, which in turn reduces the efficiency of the processing.

In some cases, the adaptation of a sequential algorithm into a parallel algorithm introduces other overheads in addition to the communications overhead. These additional overheads may involve "housekeeping chores" and imply that not all sequential algorithms are adaptable to parallel implementations [Cve87]. Also, sequential algorithms may not expose all the parallelism present in the problem [HaL82].

2.4 Other Implementations

As noted in the introduction, there have been several software implementations recently developed for parallel computers. This section briefly describes some of these implementations that use loosely-coupled MIMD computers like the Intel iPSC. Many applications have been developed at Caltech and NASA JPL in a wide range of problem areas such as high energy physics, fluid flow, astrophysics, image processing, chemistry, structural mechanics, and other areas [FoO84,Sei85, Fox81].

These applications are too numerous to describe here, but a few select applications are described, along with applications developed by researchers at other institutions.

2.4.1 Boost-Phase Track Initiation Algorithms One implementation closely related to the ones that will be developed in this thesis was developed by Gottschalk at Caltech. His implementation was developed on a version of their Cosmic Cube which is similar to the Intel iPSC. The problem involved determining the tracking of ballistic missiles in the boost phase by selecting the likely missile tracks and eliminating the unlikely or redundant tracks. His solution method used sequential algorithms in each node of the hypercube with the number of nodes a factor of 4 less than the number of targets per track. Significant speedups over sequential implementations of the same Kalman filter technique used in the parallel version were achieved [Got87].

2.4.2 Parallel Branch and Bound Mraz developed two implementations of a parallel branch-and-bound algorithm for the Intel iPSC. He solved an N-queens problem and a deadline job scheduling problem using the branch and bound technique. His method used a tree structure embedded into the hypercube interconnection network that was used to search the problem solution space [Mra86]. He reported speedups over sequential implementations of similar algorithms, however for small problem sizes, the sequential implementation performed better. This appeared to be caused by several factors, one which involved problem partition size. The other factor was related to synchronization of the tasks within the hypercube, which reduced the amount of parallelism achievable. As the size of the problems were increased, the speedup and efficiency of the parallel implementations showed good improvement. The results of this thesis point out the important effects problem partitioning and processor communications have on the overall performance.

2.4.3 The Traveling Salesman Problem This implementation was also developed at Caltech on one of their Cosmic Cube parallel computers. The traveling sales-

man problem is a classic optimization problem that has applications in areas such as circuit layout, VLSI design, resource allocation, and logistical problems [FeK85]. The basic problem is to find the shortest tour for a traveling salesman who must, for the least cost, visit a number of cities only once. The solution space of this problem grows factorially as the number of cities are increased linearly because of the number of possible routes the salesman could take. The solution method utilized by Felten and his associates was a statistical mechanics technique known as simulated annealing. A mesh structure was embedded into the hypercube network in order to match the structure of the simulated annealing algorithm. This implementation exhibited speedups over sequential implementations ranging from 1.92 using two processors to 54.92 using sixty-four processors. These speedups are not ideal, but represent significant reductions in the processing times required to solve this important optimization problem.

2.4.4 Gaussian Elimination Gaussian elimination is a computationally intensive method used to solve dense linear systems that requires manipulations of the rows and columns of large matrices [Saa86]. Saad examined several methods of mapping solutions to this problem onto the Intel iPSC computer. He found through computational experiments that this particular problem was best solved using a grid structure embedded into the hypercube network. The use of a piplining technique combined with the grid algorithms produced the lowest amount of communications between processors, which was pointed out in Section 2.3.2 as the most important overhead to reduce.

2.4.5 Vision Algorithms A hypercube implementation that applies image processing techniques to printed circuit inspection was accomplished by Mudge and Abdel-Rahman. They used a gray-code scheme similar to a Karnaugh map to partition and assign portions of an image to separate processors in a 128-processor NCUBE hypercube computer [MuA87]. Their problem was to process the image of

a printed circuit under inspection in order to extract certain features and compare them with a template of the correct image. If discrepancies were detected between the template and the image under inspection, the printed circuit was rejected as faulty. The solution of this problem required the processing of approximately 10 Mbytes of data in a few seconds. Although they used a 128-processor machine to obtain their experimental results, they predicted that 40 frames of 512 x 512 1-byte images could be completely processed in less than three seconds using a 1024-processor version of the same NCUBE computer. Two problems they encountered were computational overheads in algorithms and in communications, which were cited in Section 2.3 as potential problems with parallel implementations.

2.5 Summary

This chapter presented a brief discussion of parallel-processor architectures and an overview of Intel's iPSC MIMD computer. Techniques for developing applications for machines similar to the Intel iPSC were discussed, with particular emphasis on problem partitioning and interprocessor communications. A few of the many recent implementations on parallel MIMD computers were presented and some of the problems associated with those implementations were noted. Chapter 3 begins the process of developing parallel weapon-target assignment algorithms by examining sequential assignment algorithms. Mapping techniques introduced in this chapter are expanded for use in developing parallel implementations of assignment algorithms utilizing as many features as possible from these sequential algorithms.

3. *Development of the Parallel Assignment Algorithms*

In this chapter, a parallel weapon-target assignment algorithm is developed for implementation on the Intel iPSC computer using the techniques presented in Chapter 2. First, a formal mathematical definition of the assignment problem is given. A background on research conducted during the past three decades on different solutions to the assignment problem is then presented. The general classes of assignment algorithms that have emerged from this research are described, followed by a detailed analysis of several candidate sequential assignment algorithms with the goal of selecting one of these algorithms for parallelization. Next, different techniques for performing parallel search of a problem solution space are explored. In the final section of this chapter, a parallel search technique and one of the sequential algorithms are selected for use in the parallel assignment algorithms. This chapter concludes with a summary of the parallel algorithms developed and their implications on the remainder of this research.

3.1 The Assignment Problem

In Chapter 1, optimum assignment was characterized as a problem whose solution time-space complexity increases factorially with a linear increase in the number of resources and requesters [Chu57]. There are several variations in the details of the how the assignment problem is stated. In some instances, it is considered a special case of the transportation problem, where there are several resources at each source of supply and multiple requests for those resources at each sink. The assignment problem addresses a special case of the transportation problem where there is only one instance of a resource at each source and only one instance of that resource is required by each requester. The transportation problem itself is a special case of a general, single-objective, linear programming problem [Ign82].

3.1.1 History Research on finding faster and more efficient solutions to the assignment problem has a long history, beginning with graph theoretical work presented by Hungarian mathematicians Kõnig and Egerváry in 1931. More recent developments were accomplished by Dantzig, Flood, Von Neumann, and Kuhn in the 1950's [Chu57]. The programming methods and algorithms developed in the 1950's form the basis for much of the work that has been done on assignment problem solutions up to the present [MaN86]. Various modifications to these original assignment algorithms have been made in an effort to enhance their execution speed and efficiency on modern digital computers [McG83, CaT80, BaG77, Hun83, Ber81, GlK74, Hat75, SrT73, MaN86].

3.1.2 Statement of the Assignment Problem The assignment problem can be stated in words as: Given a number of resources and a number of requesters of those resources, and given the profit or usefulness of each resource to each requester in the form of a rating matrix where element a_{ij} is the profit of assigning resource i to requester j, the problem is to assign each resource to one and only one requester in a way that a given measure of effectiveness is optimized [Chu57]. Mathematically, the assignment problem can be stated as follows:

Given an n^2 rating matrix

$$A = \|a_{ij}\|,\ a_{ij} \geq 0\ for\ i,j = 1,2,\ldots,n\ (n \geq 3) \qquad (3-1)$$

Find an n^2 assignment matrix $X = \|x_{ij}\|$ such that

$$x_{ij} = \begin{cases} 1 & \text{if resource } i \text{ is assigned to requester } j \\ 0 & \text{otherwise} \end{cases} \qquad (3-2)$$

$$\sum_{i=1}^{n} x_{ij} = \sum_{j=1}^{n} x_{ij} = 1 \qquad (3-3)$$

$$T = \sum_{i,j} a_{ij} x_{ij} = minimum \qquad (3-4)$$

The conditions of Equations 3-2 and 3-3 specify that each row and column of matrix X will contain one element with a value of 1 and all other elements will be zero [Chu57]. The requirement of square matrices at first appears to limit the problem to cases where the number of resources equals the number of requesters. But situations where they are not equal can also be solved by adding "dummy" resources and requesters to make matrix A square. The associated rating or cost of these added matrix elements should be set to zero so that they will not be included in the final assignment solution. Other more efficient methods of handling this unequal situation have also been devised [BoL71a].

3.2 Sequential Assignment Algorithms

As stated in Section 3.1.1, much of the development of assignment algorithms over the past three decades has been based on the research accomplished in the 1950's by Dantzig, Flood, Von Neumann, and Kuhn. Other methods developed in the study of network flow have provided additional means of solving the assignment problem [Smi82]. Two basic approaches, a simplex-based transportation method and the Hungarian method, have emerged as the most popular means of solving the assignment problem primarily because of their simplicity and ease of implementation [Hat75, GlK74, MaN86]. Because of limited time and space, all of the many different assignment algorithms are not covered in detail. Instead, brief summaries of each are presented in this section. Then, in the following section, the transportation and the Hungarian methods for solving the assignment problem are analyzed. A detailed presentation and an example problem of both methods are presented in the Appendices to illustrate how the algorithms operate.

3.2.1 The Simplex Method The simplex method, developed by Dantzig, is a general approach that can be used to solve most all single-objective, linear programming problems. The basic approach of the simplex method is to start with a feasible solution to a problem and improve upon this solution in a step-by-step fashion un-

til an optimum solution is reached [Kre68]. A feasible solution means that all the constraints placed on the optimization (minimization or maximization) in the original problem statement are satisfied. In terms of the assignment problem, a feasible solution would be one where each resource is assigned to a different requester.

The type of problem most easily solved by the simplex method is one where there is a single objective function that is to be maximized or minimized, subject to constraints which are stated in the form of a system of linear equations. Additional variables, called slack variables, are added to this system of equations to aid in converging on the optimal solution. During the course of the solution, there are two sets of variables. One set is called *basic* and consists of variables that have been incorporated into the present version of the solution. The other set of variables is called *non-basic* and is comprised of variables not incorporated into the solution. Variables are modified and exchanged between the basic and non-basic sets one at a time until conditions indicate that an optimal solution has been reached. One of the primary disadvantages of the general simplex method is that the solution it provides is not integer-valued. Modified versions of the general simplex method have been developed to provide integer solutions, but they are somewhat less efficient [Ign82].

Because the simplex method is a general approach, specialized versions of it have been developed to solve specific problems. Different rules are adapted for selecting variables to enter the basic set and vary according to the type problem being solved. One example of a specialized version is the transportation method which will be described next.

3.2.2 The Transportation Method The transportation problem originated from studies made to improve the efficiency of utilizing available transport capacity in the railway and trucking industries. An example of this type problem is minimization of the cost of moving empty freight cars from their present locations to other locations where they can be used to transport goods [Chu57]. The transportation problem

existed prior to the development of the simplex method. However, efficient solutions had not been developed for it until the techniques of the simplex method were applied [Ign82]. As stated earlier, the assignment problem is a special case of the transportation problem. The transportation method uses a cost or rating matrix to represent the problem similar to the one described in the assignment problem statement. An additional row and column is added to the rating matrix to represent the number of resources available at each source and the number of requests for those resources at each sink. In order to use the transportation method to solve the assignment problem, all values in this additional column and row must be set to one.

The basic steps of the transportation method are similar to the general simplex, although they are somewhat obscured by the matrix representation of the problem. Many of the computations that would be normally be required by the simplex method are avoided by exploiting this matrix representation and using a somewhat simpler approach [Kre68, Ign82]. There are two phases to the transportation technique. The first phase generates a basic feasible solution to satisfy all the problem constraints (i.e., make initial assignments of all resources to all requesters). The second phase consists of determining whether or not the initial solution can be improved. If not, the algorithm terminates. Otherwise, the current assignment is reshuffled to improve the value of the objective function. This reshuffling is analogous to the exchange of basic and non-basic variables in the simplex technique [Ign82]. When the solution obtained in this manner cannot be improved upon, or if it is found to be unbounded, then the algorithm terminates. The exact steps of the transportation algorithm are presented in Appendix A. The next section describes another modification to the simplex method.

3.2.3 The Alternating Basis Algorithm The Alternating Basis (AB) algorithm is a modification to the simplex method which avoids the unnecessary inspection of alternative feasible solutions [BaG77]. It was presented in 1977 by Barr, Glover, and Klingman in an effort to reduce the storage requirements and compu-

tational inefficiencies of using the simplex method to solve the assignment problem. Their approach uses a rooted tree graphical representation of the problem where each node in the tree corresponds to either a source or a destination. The nodes are connected by arcs which are assigned a value of 1 if the two nodes are to be "assigned" to each other and 0 otherwise. The "alternating" part of the algorithm's name stems from the alternating manner in which the 0-arcs and 1-arcs are distributed in the tree structure. By restricting the tree structure to the "alternating path" as it is referred to in their paper, degenerate solutions that would normally be considered by the general simplex method are avoided and the efficiency of the computations is increased. The feasible solutions or bases that are considered are incrementally improved in a step-by-step manner exactly as in the general simplex method.

Some computational comparisons of the AB algorithm against other implementations of simplex-based algorithms were made by Barr and his colleagues. The results showed that the AB algorithm was approximately 15% faster than the closest competitor [BaG77]. The number of basic and non-basic variable exchanges was reduced by as much as 25% over the other methods in the comparison. These performance figures indicate that improved performance of the simplex method is strongly dependent on the rules for selecting variables to enter the basic solution set.

3.2.4 The Hungarian Method Kuhn presented a paper in 1955 describing a method for solving the assignment problem which he titled "The Hungarian Method for the Assignment Problem" [Kuh55]. The overall scheme of the Hungarian method is based on a theorem proved by the Hungarian mathematicians König and Egerváry [Chu57]. Their theorem involves covering, or including in sets, the elements of a matrix which belong to one of two distinct classes. In the Hungarian method, these two classes are formed by simple subtractions from members of the rating matrix which yield null elements and non-null elements. The *minimum* number of coverings, referred to as *lines* by Kuhn, that include all these null elements is equal to the *maximum* number of elements in that class. The Hungarian method provides min-

imum or maximum cost assignments if the resulting null elements are *independent*. Independent means that no other null elements occur in the same row or column. This restriction is analogous to permitting the assignment of each resource to only one requester and vice versa. When the number of covering lines equals the number of resources to be assigned, then there exists in the set of covered null elements at least one optimal assignment of all the resources. The method works on the principle of selectively reducing all elements in a row or column by the same amount and locating independent positions in the matrix that first become null. These independent null positions correspond to minimum or maximum cost assignments. A detailed presentation of the Hungarian method and an illustrative example problem are included in Appendix B. Several modifications of the Hungarian method have been made since its introduction. Some of these modifications are presented in the next section.

3.2.5 Modifications to the Hungarian Method Since the Hungarian is one of the more popular algorithms for solving the assignment problem, it has received the most attention by researchers who desired to improve its efficiency. Carpaneto and Toth published an improved version of the Hungarian method in 1980 which reduces the amount of time required to locate the zero elements and the unexplored rows of the current cost matrix. They used pointers to accomplish this improvement, which also reduced the storage requirements of the implementation. Another improvement they made over the original Hungarian method was to modify the choice of the initial assignment solution. Computational experiments showed that the modified algorithm outperformed other implementations of the Hungarian method for densely populated rating matrices.

Bersekas performed a more drastic modification to the Hungarian method. He changed the way that the cost of assignments were incremented during the course of the algorithm, which resulted in a faster convergence on the optimal assignment. He called his method *outpricing*, which effectively reduces the row operations required on

the rating matrix and results in solving a problem of a smaller dimension. The basic concept behind his method involves cooperative bidding where requesters attempt to outbid each other for the resources to be assigned [Ber81].

Bourgeois and Lasalle modified the Munkres version of the Hungarian method to include more efficient means for solving assignment problems that are not square (i.e., the resources do not equal the requesters) [BoL71a, BoL71b, Mun57]. They present a proof that uses two submatrices, one consisting of the real and the other consisting of the dummy resources or requesters. They argue that if the cost of the assignments in the dummy submatrix are set high enough, then all of the real resources will be assigned first. They conclude that the addition of the dummy elements is not necessary. They present an algorithm and computational comparisons with the original Hungarian method to show that their method is a performance improvement, especially when dealing with rectangular cost matrices.

3.2.6 The Branch and Bound Algorithm The Branch and Bound algorithm for the assignment problem was presented by Land and Dorg in 1960. It is a technique where a small portion of the many possible combinations of assignments are selected and an objective function is evaluated subject to certain bounds or restrictions. The basic approach is to obtain an optimal value of the objective function that lies between upper and lower bounds. The objective function's value cannot be less than the lower bound. The upper bound is normally the value of the best feasible solution obtained thus far in the current algorithm iteration. The algorithm terminates when it can be determined that there is no lower bound less than the current upper bound [Ign 82]. The branching part of the algorithm partitions the solution space into smaller, mutually exclusive subsets. The lower bounds associated with each subset are calculated and compared with the current value of the upper bound. If the lower bounds are not less than the current upper bound, then the subsets are not partitioned any further since no better solution could be obtained by branching. This branching process is repeated until all possible subsets have been formed or

until none of the lower bounds are less than the current best feasible solution.

3.2.7 The Out-of-Kilter Algorithm The Out-of-Kilter algorithm resulted from the study of optimal network flow and was presented by Ford and Fulkerson in 1961 [Dan63]. Its original application was to find either minimal or maximal flow through a network. However, applications have been found in other areas, including the solution of transportation and assignment problems [Smi82]. The network is represented as a directed graph where the nodes correspond to locations and the arcs represent links between different locations. Associated with each arc is a cost per unit flow c_{ij} through the arc which connects node i and node j. The actual flow through the arc is x_{ij}. The out-of-kilter algorithm is based on the conservation of flow at all nodes of the network: what flows into a node must flow out. The conservation of flow at node i is represented by a multiplier π_i. The π multipliers of two nodes are combined with the c_{ij} of the arc connecting these nodes to produce the flow value x_{ij}. Upper and lower bounds on the flow x_{ij} can also be imposed, but are not needed when the algorithm is used for solving the assignment problem [Smi82]. The points $(x_{ij}, c_{ij} + \pi_i - \pi_j)$ are plotted to determine if they fall on a "kilter line" which is graphically derived from the upper and lower flow bounds, and the conservation of flow multipliers for each node.

The algorithm first assigns initial flows to each arc and then searches for an arc that lies off the kilter line, which is termed being "out-of-kilter." An arc is brought into kilter by adjusting all flows in the network from the source to the destination linked by this selected arc. This process is repeated until all the arcs are brought into kilter. The assignment problem can be solved using the out-of-kilter algorithm by representing the rating matrix as a graph where the nodes correspond to the resources and requesters, and the value of the arcs correspond to the cost of assigning resource i to requester j. The upper bound on flow through each arc must be set to infinity (or a "large" number) and the lower bound to zero. Then an additional node must be added that is linked to each resource and requester node. The arcs associated with

this new node are assigned a zero cost, and an upper and lower bound of one. Then the out-of-kilter algorithm is used to find the minimal-cost feasible flow through this network. The optimal assignment is reached when the number of resources flowing through the new node is equal to the number of resources to be assigned [Smi82].

3.3 Evaluation of Candidate Algorithms

In this section, two sequential algorithms for solving the assignment problem are analyzed. First, the simplex-based transportation method is evaluated. The detailed steps of the transportation method and a small example problem are worked out and included in Appendix A to illustrate the algorithm. The Hungarian method for the assignment problem is analyzed next. A detailed explanation of the Hungarian method and an example problem using the same cost matrix data as the transportation method example are also included in Appendix B so that some comparisons of the two algorithms can be made.

3.3.1 The Transportation Method Many variations of Dantzig's simplex method have been devised in order to solve specific problems. One modification to the simplex method was made by Dantzig himself and it was done to allow a simpler solution to the transportation and assignment problems [Chu57, Dan63, Ign82]. The basic approach of the simplex method was described in Section 3.2.1 and will not be repeated here. A brief overview of the transportation method was given in Section 3.2.2 and an expansion of that overview is presented here.

The transportation method utilizes a table representation similar to the cost matrix described in the assignment problem statement of Section 3.1 where the c_{ij} elements represent the cost of assigning resource i to requester j. Some modifications are needed which involve adding another row and column, and providing additional space for maintaining some intermediate calculations. Also, elements x_{ij} of the assignment matrix are incorporated into this tabular representation in or-

der to facilitate improvement on non-optimal assignments during the course of the algorithm. An example of the table representation is shown in Table 3-1.

Table 3-1. Example Transportation Table

requester → / resource ↓	1	2	3	4	a_i ↓
1	x_{11} c_{11}	x_{12} c_{12}	x_{13} c_{13}	x_{14} c_{14}	1
2	x_{21} c_{21}	x_{22} c_{22}	x_{23} c_{23}	x_{24} c_{24}	1
3	x_{31} c_{31}	x_{32} c_{32}	x_{33} c_{33}	x_{34} c_{34}	1
4	x_{41} c_{41}	x_{42} c_{42}	x_{43} c_{43}	x_{44} c_{44}	1
b_j →	1	1	1	1	4

In cases where the number of resources does not equal the number of requests, "dummy" resource rows or requester columns with zero cost elements must be added to the above tabular representation. The a_i and b_j entries for

these additional rows or columns must be sufficient to balance the number of resources and requesters [Ign82].

There are two phases to the transportation method. The first phase is to formulate the initial basic feasible solution. The second phase checks the initial solution for optimality and incrementally improves upon it until it is optimal. The most difficult portion of the transportation algorithm is the search for the θ-paths, explained in Appendix A, that are required for the assignment of the ϵ allocations and for the exchange of basic and non-basic variables. For large problem sizes, these operations would tend to dominate the computation time. Another potential bottleneck whose details are explained in Appendix A is the satisfaction of the Δ_{ij} relationship where the R_i and K_j values are determined for assigned cells and the

values of the Δ_{ij}'s are computed for the unassigned cells. There does not seem to be any obvious shortcuts to reduce the requirements of these θ-path and Δ_{ij} operations.

In order to allow a comparison with the Hungarian method, the computational time complexity of the transportation method needs to be estimated. There are many assumptions which can be made that will affect the complexity estimate. In order to simplify the estimate made here, the operations that will be considered are scanning a row or column, adding or subtracting from a row or column, covering or marking a line, and searching for and modifying an element of the matrix. The operations carried out on the entire matrix will be the most costly, while simple operations on single variables are the least costly. The worst case scenario is assumed to be the situation where each iteration of the algorithm adds one additional member to the final solution. The following discussion is based on the solution of an $n \times n$ matrix.

Referring to Appendix A, Step 1-1 will be considered the overhead step required for both algorithms and not considered here. Steps 1-2 and 1-3 will require n operations to locate and modify the appropriate elements. In Steps 1-4 and 1-5, all initial unassigned cells are independent, so the object is to choose the $n-1$ least cost cells. This will require scanning the n rows $n-1$ times and making n modifications to the appropriate variables to mark the positions. The total number of operations for these steps are $n + n(n-1)$ or $2n + n^2$.

The next significant operations occur in Step 2-3 where the Δ_{ij} equation must be solved for the $n + n - 1$ members of the solution set. Step 2-4 requires the calculation and assignment of values to all elements of the matrix that are not part of the solution set or $2(n^2 - (2n-1))$ operations. The entire matrix must be scanned in Step 2-5, requiring n row scans. Step 2-6 in practice could be combined with Step 2-5, so another matrix scan will not be included. The operations required to construct the θ-path are more difficult to estimate. In the worst case, all the members of the current solution set would be included in the θ-path. The scans of rows and columns to determine the path direction and the determination of the θ assignment

changes will require $2(2n-1)+(2n-1)$ operations. For Step 2-9, the worst case would be to need $n-1$ ϵ assignments. These ϵ assignments would each require at most n row scans to find the minimum element, and three row scans, three column scans, and one assignment for each of the $n-1$ cells found. Steps 2-2 through 2-9 would need to be repeated $n-1$ times for the worst case scenario assumed. The total estimated operations required are:

$$operations = n + (2n + n^2) + (n-1)(3n^2 + 12n - 12) \qquad (3-5)$$

Equation 3-5 simplifies to the following expression:

$$operations = 3n^3 + 10n^2 - 21n + 12 \qquad (3-6)$$

As a result, the transportation algorithm is $O(3n^3)$.

3.3.2 The Hungarian Method Kuhn presents a rigorous mathematical proof of the theory behind his Hungarian method, which is primarily based on one main theorem and an important property of matrices related to set theory. This theorem, proved by König and generalized by Egerváry is:

> If the elements of a matrix are divided into two classes by a property R, then the minimum number of lines that contain all the elements with property R is equal to the maximum number of elements with the property R, with no two on the same line [Chu57].

The reference to a line means a row or column of a matrix. The restriction of no two elements on the same line will be used in the Hungarian method as a means to make the optimal assignment.

The important property of matrices presented by Von Neumann is:

> Given a cost matrix $A = \|a_{ij}\|$, if another matrix $B = \|b_{ij}\|$ is formed where $b_{ij} = a_{ij} - u_i - v_j$ and where u_i and v_j are arbitrary constants, the solution of A is identical to that of B [Chu57].

This property says that if all elements in a row or column are increased or decreased by the same amount, then an equivalent assignment can still be made using the modified elements. The important roles of the theorem and property are made clearer in the presentation of the algorithm in Appendix B.

The general approach of the Hungarian method involves searching the rating matrix for the minimum values in each row or column. These minimum row or column values are subtracted from each element of the rating matrix to form a new matrix that will contain a certain number of zero elements. These zero elements form one class and the non-zero elements form the other of the two classes required by König in his theorem. If the minimum number of lines that cover all the null (i.e., zero) elements is equal to the number of resources to be assigned, then the optimal assignment is contained in this set of null elements. The method of obtaining an optimal assignment from these null elements is illustrated by an example problem in Appendix B.

Now, a few comments on the computational aspects of this algorithm. Referring to Appendix B, the Hungarian method requires extensive scanning of the rows and columns of the rating matrix, which can be time consuming for large problems. Some researchers have developed methods to reduce the amount of scanning in their implementations of the Hungarian method [CaT80, McG83]. This scanning is the major drawback to the Hungarian method. Otherwise, the operations required to implement the algorithm are straightforward. Typical operations are additions, subtractions, and comparisons.

As in the transportation algorithm presentation, the computational complexity of the Hungarian method also needs to be estimated so that the more efficient algorithm can be chosen for the parallel implementations. The same operations will be considered in this case as in the previous analysis for an $n \times n$ cost matrix. Beginning with Step 1 of the Hungarian method, the location of the minimum element in each row requires n row scans and the subtraction of the minimum element from each row

element will require an additional n^2 operations. These operations are repeated with the columns, so the worst case number of operations for this step is $2(n + n^2)$. For Step 2, locating the row with one null element may require n row scans and $n - 1$ operations to cross out other possible column null elements. In the worst case, only one such element would be found per iteration of the algorithm. Step 3 requires the same number of operations as Step 2. However either Step 2 or Step 3 would be performed, but not both. Steps 4.1 through 4.3 depend on the number of rows and columns already "marked" which corresponds to the number of assignments made in the current state of the solution. The operations required would be n row scans and $n - m$ row markings where m is the number of assignments yet to be made. Step 4.2 will require scanning n columns and marking at most m columns. Step 4.3 requires another n row scans and possibly marking m rows. The total operations for Steps 4.1 through 4.3 are $n + (n - m) + n + m + n + m = 4n + m$.

The next significant operations occur in Step 4.5 where at most n rows or columns will need to be marked. Step 5 will vary in the number of operations in each iteration, but the entire matrix will need to be scanned and each element will be either subtracted from, added to, or left the same depending on the location of the marked rows and columns. The matrix scan will require n row searches and the operations on each element will need at most n^2 steps. Step 5 operations total $n+n^2$. With the exception of Step 1, the Hungarian method will require $n - 1$ iterations to solve an $n \times n$ assignment problem if only one assignment is made during each iteration. There are other situations that may require more steps, but they are not easily estimated. The estimated total number of steps for the Hungarian method is as follows:

$$operations = 2n + n^2 + \sum_{m=1}^{n-1} (n(n-1) + 4n + m + n + n^2) \qquad (3-7)$$

Simplifyiing the expression yields the following estimate for the number of operations required for the Hungarian method.

$$operations = 2n^3 + 3.5n^2 + 2.5n \qquad (3-8)$$

The Hungarian method is also an $O(n^3)$ algorithm, but the coefficients of the expression are less than the transportation method. The implications of these analyses will be discussed in Section 3.5.

3.4 Parallel Combination Strategies

There are many ways to combine the solutions to several subproblems into an overall problem solution [Qui87]. This section examines three techniques that have been developed to perform this important task. The names of these techniques are more commonly recognized as those of sequential algorithms, but they have been recently developed into high-level, *parallel* strategies for solving problems involving combinatorial search [HoZ83, WaL85, Qui87]. The assignment problem belongs to this class of combinatorial search problems, defined by Wah as the process of finding "one or more optimal or suboptimal solutions in a defined problem space" [WaL85]. The objective of this section is to describe and evaluate these parallel combination techniques. Selection of the high-level, parallel combination strategy to be used in combination with the selected node process algorithm is made in Section 3.5.

3.4.1 Branch and Bound The basic approach of the branch-and-bound technique is the systematic search of an OR-tree representation of the problem solution space. The branch-and-bound technique begins with an initial problem and some objective function which must be either minimized or maximized. It first attempts to solve the problem directly. If this is not possible because the problem is too large to be solved in a reasonably short time, then the problem is divided into smaller subproblems. With each subproblem, constraints in the form of upper and lower

bounds are included. This process continues until all the subproblems have been either completely decomposed and solved or the problem is shown to be unbounded [Qui87]. In a parallel branch-and-bound technique, many of the functions involving decomposing, solving subproblems, and evaluating constraints can be done in parallel [WaL85]. In a multiprocessor, the decomposed subproblems are each assigned to individual processors for parallel solution. A majority of the individual processors run identical node process programs. In most cases, a centralized controller, hosted on one or more processors, is used to expand the problem nodes to be examined and determine the conditions for terminating the overall process.

Lai and Sahni examined some anomalies in parallel branch-and-bound algorithms. They observed that theoretically, faster speedup is possible with a smaller number of processors. Experimental results with a parallel implementation to solve the 0-1 knapsack problem confirmed the theories they presented, although they commented that in practice the anomalies would rarely show up, except for small problem sizes. Mraz also encountered the same type anomaly in his parallel branch-and-bound implementation of the N-queens problem solution on the Intel hypercube. In his results for the 8-queens problem, 16 processors solved the problem in 1.1 seconds while 32 processors required 2.2 seconds [Mra86]. This indicates that there are significant overheads involved with implementing branch-and-bound techniques in a parallel environment.

3.4.2 Alpha-Beta Search The alpha-beta method involves the search of an AND/OR tree representation of the problem solution space. Search of an AND/OR tree is more complicated because it combines the techniques of branch-and-bound just described and divide-and-conquer which will be presented in the next section. Alpha-beta is a method usually employed in the solution of two-person zero-sum games like chess and checkers [Qui87].

The basic approach of the alpha-beta search is to consider the present state

of the problem solution, evaluate a number of possible alternative decisions, and then incorporate those alternative decisions that result in the most advantageous solution to the present problem. Two parameters, α and β, define a search "window" which is used to prune subtrees from the solution tree that do not contribute to optimal solutions. Parallel alpha-beta algorithms typically assign different windows to each processor so that faster and deeper searches of the AND/OR tree can be accomplished [SeB82]. One problem with parallelization of the alpha-beta search is that extensive communications must be used between processors to update the search window. If communications are reduced or eliminated, the result is other overheads related to processors needlessly searching through nodes of the tree determined not optimal by another node. There is a tradeoff between reducing communications and processing efficiency in the parallel implementation of the alpha-beta search method.

3.4.3 Divide-and-Conquer Unlike branch-and-bound or alpha-beta strategies, the divide-and-conquer strategy searches an AND tree representation of the problem solution space [Qui87]. Every subproblem solution is actually a part of the overall solution, which differs from the other search techniques where many subproblem solutions are discarded. Divide-and-conquer, as its name implies, divides a problem into smaller subproblems that can be solved faster and easier than the larger, overall problem. Once all of these subproblems are solved, the results are combined to form the solution to the original problem [HoZ83]. Parallel divide-and-conquer depends on the node processes to determine the feasibility or optimality of the subproblem solution.

An important factor in the performance of the divide-and-conquer search is the "granularity of parallelism" which is simply the minimum overall problem partition size [Wal85]. Problem partitioning was emphasized in Chapter 2 as an important consideration in mapping problem solutions onto parallel computers. Another consideration in the parallel implementation of divide-and-conquer is the processor utilization. The three phases of parallel divide-and-conquer are start-up, computation,

and wind-down [WaL85]. In the start-up phase, the initial problem is partitioned and the resulting subproblems are sent to the individual processors. During the computation phase, the processor utilization is typically very good. However, during the wind-down phase, many processors remain idle while the transferring and the combining of subproblem results occur. This results in tradeoffs between problem partition size and processor utilization. Larger partitions mean longer time spent in the computation and better processor utilization. But larger partitions also reduce the amount of parallelism and limit the speedup possible over sequential algorithms.

An advantage of the divide-and-conquer over the other techniques is that interprocessor communications can be very minimal during the computational phase without any performance degradation. This is, of course, dependent on the type of problem being solved. In the wind-down phase, transferring of subproblem results to be combined into the overall solution can be viewed as communications. However, these communications do not interfere with the process running in the individual processors because at this time, they have already terminated.

3.5 Results of the Analyses

In this section, the results of the preceeding analyses are summarized. The algorithm to be used as a node process and the parallel combination technique to be used will be selected. The tentative form of interprocessor communications is then be devised. A more definitive communications protocol is established in the following chapter where the actual implementation process is described. The algorithm, the search technique, and the interproceSssor communications are used as a basis for completing the implementation of the parallel assignment algorithms.

3.5.1 Selection of Search Technique All three search techniques presented have advantages and disadvantages. The branch and bound method employed by Lai, Sahni, and Mraz exhibits some anomalous behavior related to problem size and algorithmic overheads. The cure for this behavior was larger problem partition size,

which resulted in reduced parallelism. The alpha-beta search technique suffers from sensitivity to the amount of interprocessor communications. The efficiency of the solution space search is inversely proportional to the amount of communications. Divide-and-conquer performance is also affected by the problem partition size, but the effect is reduced processor efficiency and not additional algorithmic overheads as in the branch-and-bound method. The amount of required interprocessor communications in the divide-and-conquer method is potentially the smallest of the three search techniques because the individual node processes are relatively independent, except for the combining of subproblem results. Because the divide-and-conquer method is less sensitive to problem partition size and interprocessor communications than the other search techniques, it will be the parallel search technique employed in the implementations developed in Chapter 4.

3.5.2 Selection of Candidate Algorithm In Appendices A and B, the sequential transportation and Hungarian methods for solving the assignment problem were presented in detail. In this section, one of them is selected as a basis for the node process program. The problem areas that are considered in the selection are the algorithm's complexity, partitionability, and expected level of interprocess communications.

In an analysis and comparison of the computational complexity of simplex-based algorithms and the Hungarian method, Bertsekas says a fully dense, all integer, $N \times N$ assignment problem solution using the Hungarian method is $O(N^3)$. He further states that there is "no simplex type method with complexity as good as $O(N^3)$" [Ber81]. Some rules used for selecting entering variables in the simplex method have been shown to lead to exponentially long sequences of computations [Hun83]. These statements would tend to lead one to chose the Hungarian method over the simplex method. There are other factors, however, that must be considered before deciding on the "best" method.

Several comparisons of simplex-based and primal-dual based (i.e., Hungarian) methods of solving the assignment problem have been made [Hat75, GlK74, McG83]. There is general agreement that an efficient implementation of the Hungarian method is better overall than using the simplex method. One reason for this is that the Hungarian method is an algorithm developed and optimized especially for solving the assignment problem, while the simplex method is a more general method that can be used to solve a variety of linear programming problems which cannot be solved by the Hungarian method. Although variants of the simplex method have been developed for the assignment problem, they still encounter difficulties with examining nodes that do not lead to the optimum solution [Hat75, BaG77, Hun83]. Simplex methods are generally more suitable for the transportation problems discussed in section 3.1 where the number of non-degenerate arcs between nodes is less because of multiple resources and multiple requests by each requester. In a comparison of minimum-cost network flow problem solutions, a specialized simplex-based code was shown to outperform other codes which included a primal-dual code, of which the Hungarian is special case [GlK74]. But there, the main emphasis was on transportation-type problems rather than assignment problems where the simplex method seems to perform worse.

Both methods use similar matrix representations of the initial problem, the intermediate results, and the final solution. The partitioning of the problem representations of both methods is essentially the same because of the similar matrix representation. If the partitions of the square cost matrix are in the form of square sub-matrices, then information on both the costs of assigning to each requester a particular resource and the cost of assigning each resource to a certain requester will be incomplete. However, if the cost matrix is partitioned into "strips," each processor will either have complete cost information on the assignment of a group of resources to all requesters or a group of requesters to all resources. The availability of complete cost information will have an effect on the optimality of the assignments

made and the amount of assignment coordination required by the individual node processes. If the problem is partitioned into "strips," both methods use exactly the same technique of "dummy" variables to form the required matrix format where the number of resources must equal the number of requesters. The performance of the two algorithms will be affected in much the same manner by the partition type. The square partition appears to potentially require more interprocess communications than the rectangular strip partition.

The format and type of communications that will be used between processors are discussed in Section 3.5.3 and developed in Chapter 4. However, some assessment needs to be made of the level of communications that might be required by the transportation and Hungarian methods in order to develop a node process with minimal communications. The volume of communications will depend strongly on the partition size and type in both methods. As mentioned in the previous paragraph, the strip size will affect the amount of communications required to either obtain cost information or coordinate the assignments. Because the problem representations and solution results are similar, the level of communications is expected to differ very little between the two methods.

Because the partitionability and the communications requirements are very similar for both algorithms, the selection for use in the parallel algorithm must be based on some other criteria. Fox feels that the processes running on the individual nodes of a multiprocessor should essentially perform the same operations as the sequential version of the algorithm [Fox84, FoO84]. For this reason, it is reasonable to select the most efficient sequential algorithm for parallelization, provided that all other factors are nearly the same. The performance comparison by Bersekas of the Hungarian and simplex-based methods indicates that the Hungarian method is more efficient. In Section 3.3, the complexity analysis showed that the Hungarian method would require only 2/3 as many steps as the transportation method to solve the same size problem (i.e. the Hungarian method is potentially 33% faster). Based on

the previous discussion in this section and the complexity advantage, the Hungarian method is selected as the basis for the node process program.

3.5.3 Interprocess Communications Protocol Based on the discussion in Chapter 2, the development of a communications protocol that both minimizes the amount of communications and permits the efficient transfer of required information between processors is the most crucial aspect of developing a parallel software implementation. The analysis of the divide-and-conquer technique pointed out that the interprocessor communications were minimal in the computation phase, depending on the type problem being solved. One type of communication envisioned is that the cost of assigning a particular resource to a requester will require knowledge of the assignment cost for all the requesters. The communication of cost information could be eliminated by storing the needed information in all processors. This concept is explored further in Chapter 4. At this point in the development, it is not clear whether this method is feasible.

The assignment problem will also require some degree of communications between processors so that the assignments made by other processes can be checked to see if the same requester was assigned more than one resource. However, this communication would occur after an iteration of the assignment algorithm was complete. If conflicts are present, then a form of bidding would need to take place where the lowest cost assignment to a requester would stand and all processors that assigned other resources to the same requester would need to recompute another assignment without considering the conflicting requester. This will obviously require that after each assignment by a node processor, the individual assignments would need to be broadcast to other processors to determine if any conflicts exist. If none exist, the assignment stands. Otherwise, the bidding process would occur to resolve the conflicting assignments. There are several unanswered questions about the exact form of the interprocessor communications. But the concepts just presented should form a basis that can be further refined in the implementation process that follows.

3.5.4 The Parallel Assignment Algorithms The general scheme of the parallel assignment algorithms are based on the divide-and-conquer as the high-level parallel search strategy and the sequential Hungarian method as the node process program. Each processor in the Intel hypercube runs a version of the sequential Hungarian algorithm that has been modified to include a means of communicating with other processors and the cube manager. The exact form of the interprocessor communications is not fully defined at this point, but the majority of the communications involve the exchange of information to resolve conflicts in assignments after each complete iteration of the Hungarian algorithm in the node processors. The implementations in Chapter 4 use the general scheme described here and refine it as necessary.

3.6 Summary

This chapter has covered the development of the parallel assignment algorithm, beginning with a formal definition of the assignment problem. The major sequential algorithms developed for solving the assignment problem were briefly described, followed by a detailed presentation of the transportation and Hungarian algorithms. The Hungarian and transportation methods were compared in terms of computational complexity and suitability for parallelization. Then three methods of parallel search of a problem solution space were explored. In the concluding section, the Hungarian method was chosen as the basis for the node program to be developed in Chapter 4. The divide-and-conquer technique was chosen as the high-level parallel search method to be incorporated into the parallel assignment algorithm. And finally, the groundwork for the communications protocol was established. Chapter 4 continues the process of developing the implementation of a parallel weapon-target assignment algorithm on the Intel hypercube computer by utilizing the results of this chapter in the design and coding of the software.

4. *Implementation of the Assignment Algorithms*

This chapter utilizes the background work done in the preceeding chapters to develop implementations of parallel assignment algorithms for the Intel iPSC parallel computer. First, detailed assumptions are presented to more closely define the problem space being considered, followed by a definition of the experimental model. A brief description of a Ballistic Missile Defense (BMD) simulation program [Odo85] developed for the U.S. Army and other government agencies is then given, followed by an explanation of the method employed to generate input data for the programs developed in this research.

After fully developing the experimental model and the means to generate plausible input data, the development of two sequential assignment implementations is described. The purpose of the first sequential program, named the "sorting method," is to establish a baseline for comparison with all of the other sequential and parallel assignment algorithms. The second sequential implementation, which is based on a version of the Hungarian method developed by Bourgeois and Lassalle [BoL71a], is known as the "sequential B&L algorithm." The sequential B&L algorithm is also used for comparison with the parallel implementations. Portions of it later become an integral part of the parallel algorithms.

The development of four different parallel implementations of an assignment algorithm is presented next. Each parallel implementation uses a different level of interprocessor communications in order to allow a study of its effect on performance measures such as computation times, speedup, load balancing, and assignment costs. The description of each implementation states the objectives, outlines the development approach, defines the software modules and interfaces, describes the organization and major sections of the program, and estimates the computational complexity. This chapter concludes with a summary of the implementations developed.

4.1 The Experimental Model

In Chapter 1, a number of assumptions were stated concerning the deployment and operation of the BM/C3 battle management system. This section expands on those assumptions and states them in more detail. Then the battle management portion of the BM/C3 system being modeled is precisely defined. A background and brief description of the BMD simulation program is given in order to further define the scope and limitations of the experimental model. This section concludes with the development of a simpler program to generate data similar to the attack scenarios generated by the BMD simulation program.

4.1.1 Assumptions The main point of the assumptions stated in Section 1.6 was that this study focuses on one specific task of the battle management function. That function is the weapon-to-target assignment. All other functions related to the management of the weapons and other resources are assumed to be handled by other "modules" or components of the system. The optimal assignment of weapons cannot be accomplished unless certain information from these other modules is available. Information such as the range to the target, the type of target, the weapon-to-target impact angle, the expected impact area of the target, the status and position of all weapons, and several other factors are all needed to derive the "cost" of assigning each weapon to each potential target. The data collection and evaluation activities required to derive these individual assignment costs are assumed to be performed by the other modules and made available to the assignment module of the battle management system. The assignment module is further assumed to be memoryless. This means that each assignment iteration is based only on the current cost information provided to it and is unaffected by previous assignments. However, the assignment process may choose to allow certain weapons to remain idle for future use if the present cost of utilization is considered too high.

4.1.2 Model Definition The system model used in this research is not geared towards any specific type of weapon. "Generic" weapons are assumed to be deployed on space-based platforms orbiting the earth. These weapons are also assumed to be "single shot" in the sense that during each assignment, each weapon can be assigned to only one target. For weapons with multiple targeting capability, multiple assignments would occur over several iterations of the assignment task with other modules accounting for factors such as slew rates and retargeting capabilities. Each iteration of the assignment task is a single "snapshot" in the overall battle management process that would occur in the interception of ballistic missiles. This model is not intended to account for all factors involved with the BM/C3 task, but rather to address the major issues that affect the critical assignment task.

4.1.3 The Ballistic Missile Defense Simulation Program Several simulation programs have been developed in the past to model the development and deployment of ballistic missile defense systems [Odo85, Cur87]. One recent simulation program is the result of work performed under contract to the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army. DESE Research and Engineering was tasked to develop a software and graphics package to aid in the research and development of BMD and Anti-SATellite (ASAT) programs [Odo85]. The main objective of this project was to utilize interactive graphics to aid in assessing the performance of proposed scenarios and weapon deployments. A FORTRAN-based testbed program was written to model the significant physical parameters of the problem and generate data to be used in developing the graphics driver program written in a version of the FORTH language. The testbed program combined the results of earlier research and provided a first order simulation of the events expected to occur in a full-scale global engagement. The primary weapon system modeled in this simulation program was a combined ground-based laser and space-based relay mirror arrangement. The engagement scenarios generated were plausible because the enemy missile trajectories were calculated as originating from known missile

sites in the Soviet Union and terminating in major cities and military complexes in the United States. The altitudes and orbits of the relay mirrors were also simulated and the visibility of each mirror to the ground-based lasers were determined using three-dimensional coordinates, the rotation of the earth, orbital mechanics, weather conditions, and several other factors.

4.1.4 Method of Input Data Generation The BMD simulation program briefly described above focused on one particular type of weapon system. One component of that weapon system was a constellation of space-based relay mirrors. Certain subroutines of the program calculate the distance between a relay mirror and a target, and the incident angle of a laser directed at a particular target. These two pieces of information can also be used as basic parameters for an entirely space-based weapon system. However, the BMD program is not capable of producing more than 20 feasible weapon-to-target "links" per snapshot. This is much too low to be useful because the assignment algorithms studied in Chapter 3 would treat this small number of weapons as a trivial case.

The distance and angle data are still very useful, even though the quantity of data is insufficient. The testbed program was modified to store this information during the execution of a typical full-scale attack simulation. Representative values of the distances and angles were used as a guide to develop similar and more extensive data by means of a much simpler program. One heuristic used in the BMD simulation program to indicate a potentially good assignment was the weapon-to-target distance multiplied by the cosine of the impact angle. This produces low values for impact angles close to 90 degrees and high values for angles near zero. For minimum cost assignments, this heuristic is expected to work reasonably well for other types of directed energy weapons that are likely to be deployed.

The data generation program developed in this research uses a random number generator to produce values in the range of 1 to 2500, which correspond to the range

of values for the heuristic just described. Although the data is somewhat random, provisions were made to produce lower values in some sections of the cost matrix and higher values in others. The lower values correspond to a high probability of kill and low cost assignments. The higher values indicate low probability of kill and high cost assignments (i.e., long distances, small angles of impact). The groupings of low and high values are intended to represent groups of weapons that have similar opportunities for engaging the same targets. One additional factor that can be accounted for is the number of reentry vehicles (RV) contained in a particular booster-phase target. An individual cost value can be lowered further by a factor of the number of RVs, which make that particular target more likely to be engaged by the assignment algorithm.

4.2 Sequential Assignment Algorithm Implementations

Two sequential assignment implementations are described in this section. The first one, called the sorting method, is typical of the methods used in battle management simulation programs to assign weapons to targets and does not provide the optimal assignment [Odo85, Cur87]. The second one, called the sequential B&L algorithm, is based on a version of the Hungarian method developed by Bourgeois and Lassalle which does provide the minimum cost optimal assignment of available weapons.

4.2.1 The Sorting Method The purpose of this assignment implementation is to provide a baseline that is relatively easy to implement and can be used for comparison with more efficient assignment problem solutions. It illustrates how a commonly used, simple approach to the assignment problem solution can be very time consuming. The basic approach to this program, as the name implies, is sorting. The input data generated by the program described in Section 4.1.4 is normally stored as a rating matrix. For this application, the data is reordered into a list or one-dimensional array format to allow the cost values to be sorted in ascending order.

The row and column values associated with each cost entry are accessible through corresponding arrays. Once the list is sorted, the lowest value is selected and the associated weapon and target (row and column) are marked as "assigned." Then the next lowest cost in the list is selected and if both the weapon and the target associated with this value are also unassigned, the assignment is made. However, if either the weapon or the target is already assigned, the next lowest value in the list is examined for possible assignment and so on. This process continues until all the weapons are assigned.

At first, this method appears to offer the lowest possible overall assignment cost. However, this is not the case. In many cases, a lower cost assignment is selected for a particular weapon-target pair. This eliminates the possibility of using the same weapon or target in a later assignment which, although some of the individual assignment costs may be higher, the effect would be a lower overall cost. The results of the example problems in Appendices A and B illustrate this point. If an assignment was made using the same cost data given in the example with the sorting method, the overall cost could have ranged from the optimum on up to a value of 15, depending on how the list was sorted.

The program implementing this algorithm is actually split into two portions. The first portion is hosted on the cube manager of the Intel iPSC. Its function is to prompt the user for problem size information, access the file containing the input data, and compile post-run statistics. The other portion of the program is hosted on one of the node processors of the iPSC. The cube manager or host program sends the problem size parameters and the cost data to the node program. The node program sorts the cost data using a relatively quick Shell sort [KeR78] and performs the process previously described to make the assignments. Once the assignments are complete, the node program sends the assignments list and timing information to the host program for further processing and display. The performance results of this implementation are presented and analyzed in Chapter 5.

The basic limiting factor of this method is the sorting of the cost list. Some simulation programs use heuristic methods to reduce the size of this list in order to shorten the time required to sort the list. The Shell sort is an $O(N \times \sqrt{N})$ procedure [Fel85]. When the assignment process is performed on the sorted list, it will be at least $O(N)$ because N weapons need to be assigned. In the worst-case, N^2 operations will be required to make the assignments using the sorted list. Overall, this sorting method of solving the assignment problem appears to be $O(N^{3.5})$.

4.2.2 The Bourgeois and Lassalle Algorithm The Hungarian method, selected in Chapter 3 as the basis for the node process of the parallel assignment algorithm, can be found in many forms in the literature. The Bourgeois and Lassalle (B&L) algorithm is one variation of the Hungarian method [BoL71b]. It is chosen for implementation because it handles the case of non-square cost matrices without the addition of dummy variables mentioned in the presentation of the Hungarian method. In a realistic scenario, there will be many more targets than there are weapons. The basic operations of the B&L algorithm are the same as those illustrated in the example problem in Appendix B with the addition of some pointer arrays to keep track of certain assigned weapons and targets.

The basic approach to solving the assignment problem in this implementation is to use the B&L algorithm as a function call within the same basic framework as the sorting method program. The problem size parameters and input cost data are handled by a cube manager host program. The actual assignment is performed by a node program and the results are sent back to the host program for post-run processing. The operations required to formulate the optimal assignment are contained within the B&L algorithm and are illustrated in the example problem of Appendix B. The cost matrix, the number of weapons, and the number of targets are all supplied as parameters to the assignment function call. The function returns an array indexed from one to the number of available weapons and the total cost of the assignment. Each entry in the array is the target number to which the weapon

(the array index number) is assigned.

This program will provide a minimum-cost, optimal assignment. The overall cost of the assignment is considered when each individual assignment is made, unlike the sorting method where only the individual costs are considered. The use of the matrix representation and several arrays to keep track of potential and actual assignments of weapons and targets allow the optimal assignment to be derived. The performance improvement of this program over the sorting method is presented in Chapter 5.

The complexity of the Hungarian method was analyzed in Section 3.3.2. The complexity of the B&L algorithm is somewhat worse for square matrices because of the leading n^3 term's coefficient. For nonsquare matrices, the complexity is slightly improved. Fewer operations are required because, instead of searching and subtracting both column and row minimums, either row *or* column minimums are searched for and subtracted. In the nonsquare case with m rows and n columns where $m < n$, $m + mn$ operations are required to locate and subtract minimum values compared to the $2(n + n^2)$ operations in the pure Hungarian method where the matrix must be squared with dummy elements. Other operations in the B&L algorithm are reduced by similar factors because the extra dummy rows or columns are not needed. Overall, the total number of operations are significantly less in the worst-case where each iteration adds only one additional assignment. Instead of $n - 1$ complete iterations, only $m - 1$ iterations are required. Performing a complexity analysis similar to the one in section 3.3.2, the complexity estimate for the version of the B&L algorithm developed here is:

$$operations = 3nm^2 + 4m^2 - 2nm - m - 1 \qquad (4-1)$$

This complexity estimate will be later used in assessing the complexity of the parallel versions of the assignment algorithm.

4.3 Parallel Assignment Algorithm Implementations

In this section, four different parallel implementations of the assignment algorithm are presented. The first three programs are closely related and vary mainly in the amount of interprocessor coordination. Although all of the first three programs use the B&L algorithm function developed in Section 4.2 to perform the assignment task in parallel, the final assignments produced are not optimal Heuristics are used to reduce the number of redundant assignments and will be fully discussed in the following sections. The fourth program is an effort to implement a parallel version of the B&L algorithm whose final solution is optimal.

As studied in Chapter 2, the two major areas to be concerned with when developing parallel programs are the interprocessor communications and the problem partition size. The effects of different levels of interprocessor communications can be studied by comparing the performance of these first three programs. The type of matrix partitioning discussed in Section 3.5.2 was the strip method where entire rows of the matrix are transferred to the different processors. By partitioning in this method, each processor is responsible for a unique group of weapons. Complete cost information for assigning any of its weapons is available without initiating any communications with neighboring processors. Other forms of communications that are necessary will be discussed in the description of each program.

4.3.1 The First Level: No Communications The "first level" or level 1 parallel program is the case where there is no coordination between any of the processors in the iPSC. Each node processor works entirely independent of the other processors. Figure 4-1 illustrates the relationship between the individual processors in the cube and the cube manager. With a 5-dimension hypercube, up to 32 processors can operate in parallel on different portions of the cost matrix. The execution time is expected to be much shorter than either of the sequential implementations, however the resulting assignment will not be optimal.

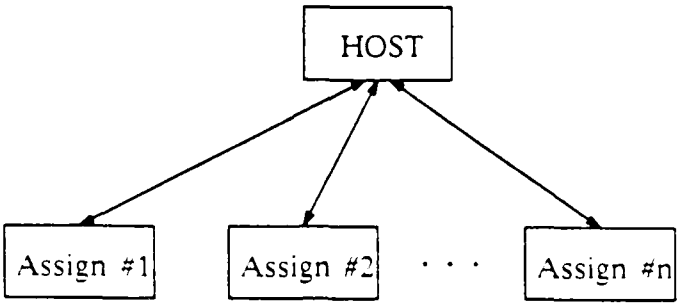


Figure 4-1. Processor Communication Paths for the First Level

The non-optimal assignment solution of this implementation results from the individual processors not communicating with each other about which targets have been assigned. As a consequence, one processor may assign a weapon to a certain target while another processor may assign a different weapon to the same target. This wastes one weapon that could have been assigned to another target. A larger number of processors will most likely result in more redundant assignments and more wasted weapons, but will yield these results much faster than could a single processor implementation. The performance evaluations in Chapter 5 address both the problem of redundancies and the tradeoffs between the speed of execution and the optimality of assignment.

This parallel implementation is simply an extension of the sequential version developed in Section 4.2.2. An identical node process is loaded into all of the processors to be utilized. The host program prompts for problem size input and reads the input cost data from an external file. But in this case, the cost matrix must be partitioned among the multiple processors. There are two situations that must be handled. One is where the rows of the cost matrix are divided evenly among the node

processors and the other is where they do not. In the uneven case, one processor will receive an odd number of rows. This will slightly affect the load balancing of the processors, but is not expected to be a significant problem. Once each processor completes its independent assignment, it waits until prompted by the host to return the assignment results and timing information.

In this implementation, the B&L algorithm is essentially running in each of the cube processors. The complexity of this B&L algorithm has already been estimated in Section 4.2.2. If the operations required to transfer the cost matrix data to the individual nodes are ignored, the complexity estimate can be derived by simply dividing the sequential B&L algorithm complexity estimate by the number of processors being used. The accuracy of this estimate is tested when the actual performance data is analyzed in the following chapter.

4.3.2 The Second Level: Partial Communications, Single Iteration This "second level" or level 2 parallel implementation introduces some coordination between the processors computing assignments for certain partitions of the cost matrix. The coordination is performed by processors designated as *controller* processors. The processors performing the assignments are known as *assign* processors. A possible processor arrangement for two partitions is illustrated in Figure 4-2. For this study, the number of controllers available is 2, 4, or 8. With 2 controllers, up to 15 assign processors may be used per controller. For 4 controllers, up to 7 assign processors and for 8 controllers either 2 or 3 assign processors per controller may be utilized. The level 2 implementation described in this section and the "third level" or level 3 program discussed in the following section both use the same basic processor arrangement.

The host program performs essentially the same function as in the first level approach. The only difference is that the host communicates with the controller processors rather than the assign processors. The partitions of the cost matrix sent

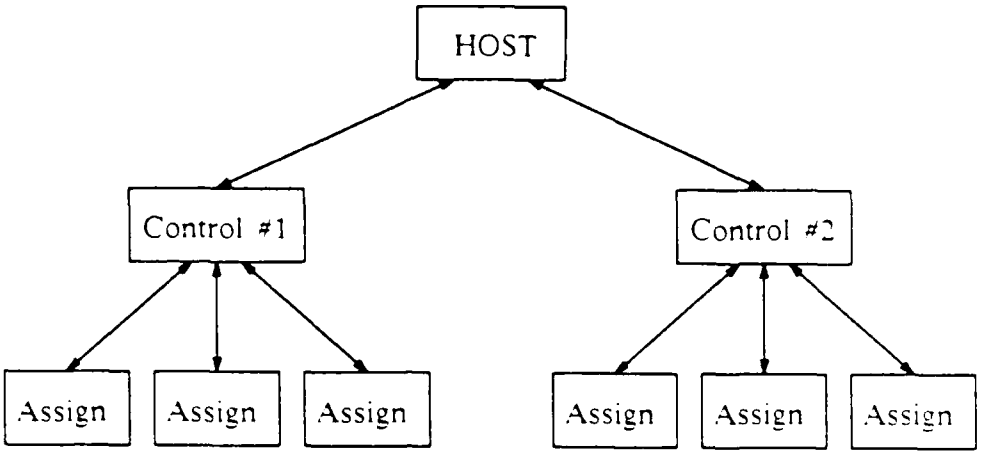


Figure 4-2. Communication Paths for the Second and Third Levels

to the controller processors are further subdivided by the controllers and sent to the appropriate assign processors. The assign processors have no direct communication with the host except when a global START command is issued from the host to signal the completion of all data distribution functions and the beginning of the actual processing. The assignments or weapon-target pairings from the assign processors are examined by their associated controllers. The controllers eliminate redundancies in the weapon-target pairings by comparing the individual costs of those that are conflicting. The controllers allow the lowest cost weapon allocations to remain and sets all the higher cost, redundantly assigned weapons to an idle state. The results and timing information are sent to the host and processed as previously described.

The results of this implementation should show an improvement over the first level approach. Fewer redundancies and lower costs are some of the expected benefits. The coordination requires extra computations that may degrade performance if a large number of redundancies occur. The final weapon-target pairings for a two controller configuration should be similar to the first level implementation using two processors.

The controller processors add a significant number of operations. For each controller, the individual assignments must be received from the assign processors and a master assignments list compiled. Then the list must be searched to identify any redundancies. A table lookup must be accomplished for each individual assignment to determine the lowest cost in the case of conflicts and to derive the overall assignment cost. The operations required for the assignment list compilation depend on the partition size. A large number of controllers allow more of the operations to be done in parallel. The worst-case for conflicts would be where every assignment from one assign processor conflicts with an assignment from another processor. The additional load from the controller is estimated to be:

$$operations = n^2/p + 2(n/p) + n/2p + n/p = (n^2 + 3.5n)/p \qquad (4-2)$$

where

n is the total number of weapons

and p is the number of controllers

This estimate just given is in addition to the complexity estimate for the level 1 implementation. The complete complexity estimate is shown in Equation 4-3.

$$operations = (3nm^2 + 4m^2 + n^2 - 2nm + 3.5n - m - 1)/p \qquad (4-3)$$

4.3.3 The Third Level: Partial Communications, Multiple Iterations The "third level" or level 3 parallel implementation increases the amount of coordination performed in the controller processors. The controller and assign processors are utilized in the same configuration as the second level approach, illustrated in Figure 4-2. Instead of idling the redundantly assigned weapons as in level 2, these weapons are made available for assignment to other targets not yet assigned.

The cost matrix is partitioned exactly as in the level 2 implementation. Each group of assign processors report to one specific controller processor. The controllers

receive assignments computed on partitions of the cost matrix from their assign processors. Each controller compiles a master assignments list. The redundancies are then determined and the lowest cost individual assignments are allowed to stand. The weapons involved in higher cost redundant assignments are entered into one list and all targets that are not assigned are entered into another list. Each controller then broadcasts its lists to all assign processors under its control. New sets of assignments are computed and sent back to the controllers which again coordinate the removal of any new redundancies. This process continues until all weapons have been assigned to a different target and all redundancies within the partitions have been eliminated. Each controller then sends its final master assignment list back to the host where it is compiled into a final assignment.

The final assignment from this implementation will also not be optimal. There may be some redundancies resulting from the assignments made in different controller partitions because there is no coordination between the controllers. However, there will be no idle weapons due to the multiple iterations performed to eliminate the redundant assignments within each controller's partition. The cost of the final assignment will tend to be higher than the optimal assignment for several reasons. When redundancies occur within a controller's partition, at least one of the final assignments made by the assign processors will not be optimal because alternative weapon-targets are always an equal or higher cost. Although redundancies are eliminated within each controller's partition, other redundancies can still possibly exist between different controllers.

The additional operations required by the controller processors are similar to the second level implementation. However, the multiple iterations required to eliminate the redundant assignments within each controller's partition are an additional source of computational overhead. In the worst-case, each iteration would only assign one of the available weapons for each assign processor. This would require n/pq iterations where n is the total number of weapons, p is the number of partitions

or controllers, and q is the number of assign processors per controller. Multiplying Equation 4-2 by the number of iterations and adding the result to Equation 4-3 yields the following expression for the complexity estimate of this level 3 implementation:

$$operations = (3nm^2 + 4m^2 + n^2 - 2nm + 3.5n - m - 1)/p + (n^3 + 3.5n^2)/qp^2 \quad (4-4)$$

The coordination process is very expensive in terms of the number of operations required. In the worst-case, it is of the approximately same order as the B&L algorithm itself. Although each iteration of the controller process requires another iteration of the B&L algorithm, the B&L algorithm is performed on subproblems of successively smaller dimensions. The dominant factor is the controller process because it requires the same number of operations on each iteration.

4.3.4 The Fourth Level: Parallel Matrix Operations The "fourth level" or level 4 implementation is a different approach from the first three parallel implementations. The program development involved studying the different operations required by the sequential B&L algorithm and identifying the operations that were the most time consuming. Then, certain operations were implemented in parallel on multiple processors.

The most time consuming operations of the algorithm were located using the timing function of the iPSC on different segments of the sequential B&L algorithm implementation described in Section 4.2.2. Several different cost matrix sizes and weapon-to-target ratios were used to determine the algorithm's performance characteristics. Three distinct segments of the sequential B&L algorithm were identified as consuming more than 75% of the processing time. Not unexpectedly, these code segments involved operations carried out on large portions of the cost matrix. Of these three code segments, one of them dominated the processing time when the weapon-to-target ratio was greater than or equal to 1:5. Because the weapon-to-target ratio

in a realistic scenario is expected to be at least 1:5 [AdW85], this particular segment of the sequential B&L algorithm was chosen for as the prime candidate for parallelization. The operations performed in this segment search for minimum row and column values, subtract the minimum values from the entire matrix, and locate the resulting independent zero elements. These operations are analogous to the first three steps of the Hungarian method description found in Appendix B. The other time-consuming code segments were not chosen for parallelization because of the higher amount of message passing that would be necessary to update various global arrays used to coordinate the refinement of the initial assignment solution.

The implementation of this program was divided into three portions. The usual host program performs the functions described in the previous implementations. There are two different node programs. One is known as the *serial* process and it performs the serial tasks of the B&L algorithm. The other node program is the *parallel* process. The multiple parallel processes are subordinate to the serial process and perform the operations identified as time consuming in the preceeding paragraph. Figure 4-3 illustrates the communication paths between processors in this implementation.

Each parallel process operates on a particular "strip" of the cost matrix. Initially, the minimum value in each row is determined and then this minimum value is subtracted from each element in that row. These row operations can be performed in parallel without any interprocessor communications. However, in the case of square matrices, the minimum elements in each column must also be determined and then subtracted. Because the row subtractions are performed on horizontal strips, no processor will contain a completely modified column. This requires that each processor search a portion of each column for minimum elements. The overall minimum element of each column is determined from the individual processor contributions by using a global operation function. The overall column minimum is then broadcast to all processors for subtraction from their segment of the column. After all minimum

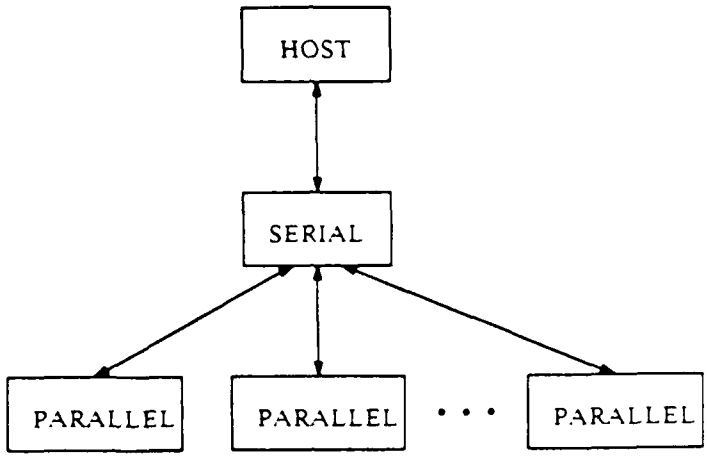


Figure 4-3. Communication Paths for the Fourth Level

elements have been subtracted, then the independent zero elements are determined and used to make the initial assignment of weapons to targets.

In the three previous parallel implementations, there is a problem with the redundant assignment of weapons to the same target. In this implementation, the problem is eliminated by coordinating the assignment process. Since each processor contains a strip of the cost matrix, the assignments will be made by using only the cost information from this strip. Two vectors, one containing the weapon number assigned to each target and the other containing the target numbers that have been assigned will be used as the means of coordination. Each strip is further subdivided into separate "windows." The parallel assignment process will require a number of iterations equal to the number of these "windows." During a parallel assignment iteration, each processor makes assignments on a different *independent* window. The term independent means that the targets being considered for assignment are not being considered by any other processor during the present iteration. The weapons are already independent by virtue of the strip method of partitioning. After each iteration, the individual assignment contributions from each processor are used to

update the two assignment vectors using a global concatenation function. Then the vectors are broadcast to each processor and the next set of independent windows are searched for possible assignments. After the last iteration of the parallel assignment, the number of weapons that have been assigned is checked. If all weapons have been assigned, then the algorithm terminates. If not, then the remainder of the program operates exactly as the sequential version explained earlier in Section 4.2.2.

The solution produced by this implementation will be the minimum-cost optimal assignment. The final results will be the same as those produced by the sequential version of the B&L algorithm described in Section 4.2.2. However, development and initial testing of this implementation indicates that it will possibly require as much or more time than the purely sequential version. The primary reason for this is the volume and frequency of interprocessor communications used to coordinate the assignments and eliminate the redundancies. Specific performance data are presented in the following chapter and comparisons are made with the other implementations.

In the nonsquare matrix case where the number of targets is greater than the number of weapons, the number of operations required at first appears to be reduced because of the multiple processors performing the operations in parallel. This holds only when there is little or no coordination required. After the row minimums have been subtracted, each of the assignment iterations on the windows described earlier require the transmission of node contributions to the serial processor, which in turn broadcasts the updated vectors back to the parallel processors. Much of the communications processing involved with the sending of messages between nodes is performed by the operating system and the number of operations involved is not easily determined. However, the sizes of the the vectors are known, so some rough estimate of the processing can be made. One vector length is equal to the number of weapons and the other is equal to the number of targets. The number of windows will be equal to the number of parallel processors utilized. At least $m + n$

operations will be required to combine the node contributions into a single vector during each iteration where m is the number of weapons and n is the number of targets. If p is the number of parallel processors, then at least $p(n+m)$ additional operations are required in the parallel implementation. The row minimum search and subtraction will also require some additional operations to transmit the cost information to the serial processor, but the actual number of operations is difficult to determine because of the message passing. After considering these additional factors, the complexity estimate of the fourth level implementation for the worst-case where only one assignment is found in the parallel segment is estimated to be as follows:

$$operations = 2nm^2+4m^2+mn(2/p+4/p^2-1)+m/p+p(m+n+1)-n-m \quad (4-5)$$

It is obvious from the complexity estimate that the parallel version of the B&L algorithm will require more operations than the level 1 implementation in the worst-case. The actual performance, using data that is not worst-case, will be examined in Chapter 5.

4.4 Summary

This chapter restated the assumptions given in Chapter 1 and provided more background on the ballistic missile simulation program used as an aid in generating input data for the programs developed. Two sequential programs were presented, one which utilized a sorting method to order the assignment costs and the other which used a modified version of the Hungarian method presented in Chapter 3 and Appendix B. Four parallel programs were described which involved different levels of interprocessor communications. The first three used the B&L algorithm code replicated in certain nodes and partitioned the cost matrix among the different processors. The fourth parallel program attempted to perform certain operations of

the B&L algorithm in parallel. The computational complexity of each implementation was estimated. In Chapter 5, regression analyses is used to determine how well the plots of predicted and actual processing times match. The relative performance of each implementation is compared in terms of speedup over single processor implementations and the optimality of the assignments.

5. *Experimental Results and Performance Analysis*

The details of implementing the sequential and parallel assignment algorithms were presented in the previous chapter. Test cases were devised that were small enough to permit hand calculation of the assignment results. After these implementations were tested with this test data to insure the assignment results were correct, a series of performance runs was made with larger cost matrices and data was collected. This chapter presents these experimental results and analyzes them according to the criteria stated in Chapter 1. The specific performance criteria are computation times, speedups, interprocessor communications, load balancing, and machine-size to problem-size relationships. This chapter is organized into three major sections. The first section defines the performance criteria and the method of data collection. The second section presents the performance results of all the implementations and evaluates the predicted complexity estimates made in Chapter 4. The last major section analyzes and compares these results according to the criteria defined in the first section. This chapter ends with a summary of the experimental results and analyses.

5.1 Testing Approach

In the engagement of defensive weapons against a full-scale, global missile attack, the defensive weapons will most likely be outnumbered by the incoming missiles. Several estimates of the ratio between defensive weapons and incoming targets (referred to as the ratio of *weapons-to-targets* from here on) have been made in the open literature [AdF85, AdW85, BoW85, DrF85]. Although predicted ratios of weapons-to-targets vary, depending on the assumptions made and the method of analysis, most estimates range from 1:1 to 1:10. Based on these estimates, cost matrix sizes corresponding to weapon-to-target ratios of 1:1, 1:5, and 1:10 were chosen for this study. The number of weapons was chosen to range from 32 to 128.

This range of weapons was selected in order to study the assignment problem on a small scale and does not represent any estimate of the number of defensive weapons that may be actually deployed.

For the experimental tests, five different cost matrices for each matrix size were generated using the program that was described in Chapter 4. Several trial runs were made with each implementation to test the variability of the processing times obtained and to establish the number of test runs needed. An analysis of the means and the variances of the processing times was performed using a statistical data analysis package known as SAS (a registered trademark of the SAS Institute, Cary, N.C.) [CoS87]. Five sets of matrices for each size were chosen as the standard number of runs because the mean processing times for the same number of processors were not found to be significantly different from each other within a 95% confidence level. The same test performed on the mean processing times obtained using different numbers of processors with the same suite of input data did show significant differences, as expected. The ANOVA (*AN*alysis *O*f the *VA*riance) procedure of SAS showed that modeling the processing times as a function of the input data (the different cost matrices) with the number of processors held constant was a very poor model. It had a probability of rejection of 0.9942. This indicates that the different input data sets do not have a significant effect on the processing times. On the other hand, if the same input data was used for different numbers of processors and the processing times were modeled as a function of the number of processors, the probability of rejecting this model was less than 0.0001. This means there is a better than 99.99% chance that the number of processors used has an effect on the processing times.

5.1.1 Performance Criteria As stated in Chapter 1 and repeated in the introduction to this chapter, there are a number of performance measures that need to be analyzed and compared for each implementation. In this section, each of these measures are briefly defined and any special considerations are explained.

The first performance measure is the computation or processing time. In se-

quential processors, this performance index is relatively simple to measure. However, in an MIMD multiple-processor machine such as the iPSC, there are many factors that affect the ease with which actual processing times can be measured. The parallel solutions to many problems involve three principle phases: start-up, computation, and wind-down [WaL85]. In this research, the start-up and wind-down phases are especially time consuming because of hardware constraints imposed by the Intel iPSC. One major constraint is that there is only one serial data channel from the cube manager to the node processors. This limits the speed of transferring initial cost data to the node processors and receiving the results from the node processors. Improved parallel I/O techniques have been implemented in, for example, the NCUBE hypercube [HaM86] and the PASM prototype [SiS84]. The start-up and wind-down times in the iPSC implementation unnecessarily bias the runtimes. As a result, they will not be included in the total processing times reported. The timing will commence when all processors have received the initial cost data and terminate when the last processor finishes its computations and is ready to return results.

One common performance measure in parallel processing is speedup (S). This index relates the time to compute a solution with one processor with the time to compute a similar solution with N processors. It is defined as follows:

$$S = T_1/T_N \qquad (5-1)$$

where

T_1 is the computation time for one processor and

T_N is the computation time for N processors

If a problem can be broken down into N independent pieces, then N processors can solve these N pieces in $1/N$th of the time required by a single processor. The T_1 times reported in this research are those obtained from using one node processor of the iPSC. Perfect speedup is N, but this is not normally achieved in practice. In some instances, certain implementations achieve *superlinear* or greater than N speedup.

There are several factors that can account for this surprising result. In this research, some of the speedups reported are superlinear. One reason for this is that the start-up and the wind-down times are not included for the reasons discussed earlier. Another reason is that although the processing times and assignment costs of sequential B&L algorithm are compared with those of the parallel versions, the algorithms being compared are very different. The sequential B&L algorithm yields the optimal overall assignment, where the parallel versions are heuristic methods designed to produce acceptable results that are *near* optimal, but not exactly optimal.

In a strict interpretation of speedup, the results of two different configurations should be the same. However, in this thesis, the term speedup will be used as one measure of performance between implementations yielding very different results. For this reason, speedup alone is not sufficient and must be taken in conjunction with other measures such as the optimality of the results or the percent effective.

Interprocessor communications were discussed in Chapter 3 as one of the more important overheads to minimize in parallel implementations. In the results that will be presented shortly, the actual time spent communicating between processors will not be explicitly shown. The method that will be used to assess the communications effect will be to compare the other performance measures of the different implementations. The increasing levels (level 1 to level 4) of implementation correspond to increasing levels of interprocessor communications. The criteria is straightforward: if higher levels of implementations perform better, then higher levels of communications are better. On the other hand, if lower levels of implementations perform better, then lower levels of communications are better. Of course there are tradeoffs between different performance characteristics. Different applications may require higher performance in one area and accept poorer performance in another area. Issues of this type are discussed further in the concluding sections of this chapter.

Load balancing is a performance measure that compares the processing times of the individual processors in a parallel system. The purpose is to determine if

all processors are performing approximately the same amount of work or if several processors remain idle while a few processors are performing a majority of the total processing. Perfect load balance at first appears to be the ultimate objective, but if the load balance is achieved solely by excessive communications between nodes, then very few useful computations are likely being performed. In later sections, specific times are not presented, but representative times are discussed and the issue of load balancing is evaluated for each parallel implementation.

The machine-size to problem-size relationship or scalability is an important measure that shows how the small-scale experimental results can be applied to larger "real world" applications. The primary means of evaluating this relationship is to first use regression analyses to determine the models that best fit the data that has been collected. Then reasonable estimates, based on these models and plots of the collected data, are made for larger problem and machine sizes. Because of the nature of some problem solution times, these estimates are subject to some error and should not be taken as absolute.

5.1.2 Method of Data Collection As explained in the introduction to this section, five sets of matrices were generated for each different matrix size. Two sequential implementations and four parallel implementations were tested. The data for the single-processor B&L algorithm is included in the level 1 data presentation. From this point on, the different parallel implementations are referred to as level 1, level 2, level 3, or level 4 corresponding to the first level, second level, and so on implementations described in Chapter 4. The sorting method implementation is referenced as the level 0 implementation.

Because five runs per matrix size were earlier shown to be statistically adequate, each implementation was tested with the same set of five matrices so that direct comparisons of computation times, speedups, and communications overhead can be made. However, some of the performance runs for the level 4 implementation

were not possible because of limitations in system buffers used to handle the internode message traffic. This occurred when the cost matrix was large and greater than 16 processors were being used. Complete data for the level 0 (sorting method) implementation is also not presented because as the matrix sizes increased, the processing times increased very rapidly.

In some instances, the 0.005 second resolution of the timing function of the iPSC affected the accuracy of the timing results. Cases where the error exceeds 10% of the reported mean of the processing times are marked with an asterisk (*) and are mainly confined to the 32-weapon cases where utilizing more than 16 processors resulted in processing times approaching the 0.005 second resolution. All derived speedups associated with these suspect processing times are also marked with an asterisk and are not considered to be accurate.

5.2 Presentation of Results

The results of all the implementations are presented in this section. It is organized into subsections that correspond to the name given to the implementation. All of the 96-weapon data for the three ratios discussed earlier are given in tabular form. The data for 32, 64, and 128 weapon evaluations are included in Appendix C.

5.2.1 Level 0 The level 0 or sorting method program was developed to provide a baseline for comparison with the other implementations. However, because of system load, complete data for all the matrix sizes was not obtained. For the largest matrix size (128×1280), the processing time was estimated to be in excess of three hours per run. The average processing times and assignment costs that were obtained are shown in Table 5-1. The *Size* column represents the product of the *Weapons* and *Targets* columns and shows the number of elements in the cost list that must be sorted. The entries in Table 5-1 are sorted according to the number of elements in the cost list. The *Cost* column shown in Table 5-1 represents the sums of the corresponding values from the cost matrix for the weapon-to-target assignments

made using the sorting method. The *% Effective* column represents the percentage of the weapons that actually killed a target. For example, in Table 5-1, the 100% effective values mean that each of the 32 weapons killed a unique target. Less than 100% effective means some of the weapons were redundantly assigned to the same target and were thus not fully utilized.

Table 5-1. Timing and Costs of the Level 0 Implementation

Weapons	Targets	Size	Time (sec)	Cost	% Effective
32	32	1024	2.352	6513.6	100.0
64	64	4096	16.581	6787.6	100.0
32	160	5120	21.693	392.0	100.0
96	96	9216	35.988	8646.4	100.0
32	320	10240	51.057	281.6	100.0
128	128	16384	101.011	9107.2	100.0
64	320	20480	132.016	560.0	100.0

There are three different weapon-to-target ratios represented in Table 5-1. As the ratio of weapons-to-targets increases from 1:1 to 1:10, the assignment costs drop by a factor of 10. This is caused by there being a larger number of lower cost individual assignments to choose from in the 1:10 case. When there are ten targets for every weapon, then there is a better chance of selecting lower cost weapon-to-target pairings than there is when the number of weapons and targets are equal. When there is an equal number of weapons and targets, certain weapon-to-target pairings are forced to be higher cost because the weapon that would have yielded a lower cost may have been previously assigned to another target. In the sorting method, no provision is made to reshuffle previously assigned weapons. In later implementations using the B&L algorithm, reshuffling of assignments is done to obtain a lower cost overall assignment.

The processing times shown in Table 5-1 reflect an increase as the size of the cost list increases. This is an understandable result. The elements of the normally used cost matrix are rearranged into a linear list so that they can be sorted into ascending order for the assignment process. As the number of elements in this

list grows, the sorting time grows accordingly. As pointed out in Chapter 4, the time required to sort this list is the dominating factor of this implementation. This explains why the processing time (Time) is more closely related to the size of the cost list than to the weapon-to-target ratio.

The complexity of this implementation was estimated in Chapter 4 to be $O(N^{3.5})$. Performing a regression analysis on these processing times resulted in the model shown in Equation 5-2.

$$processing\ \ time = 3.54 \times 10^{-7} WT^2 + 6.103 \qquad (5-2)$$

where

W is the number of weapons and

T is the number of targets

The *adjusted* R^2 coefficient produced by SAS in a regression analysis is a measure of how well the predicted and actual times match, with 1.0 being a perfect match. The adjusted R^2 coefficient between the predicted and actual processing times for this model was 0.9725. For an equal number of targets and weapons, this corresponds to $O(N^3)$, so the estimate made in Chapter 4 was somewhat pessimistic. However, the estimate was based on the number of operations expected. In Equation 5-2, the actual processing time is being modeled. There are many operating system functions and other lower level instructions being executed for each operation estimated. The relationship between high-level operations and these lower-level operations is difficult to determine. However, a relationship does appear to exist. The processing times and the estimated number of operations were tested for correlation and the Pearson correlation coefficient was found to be significant to better than a 95% confidence level.

5.2.2 Level 1 The level 1 implementation is the first parallel implementation where there are no communications between any of the processors in the hypercube.

The mean processing times and related speedups over both the single-processor B&L algorithm ($S_{B\&L}$) and the level 0 implementation (S_{Sort}) for the 96-weapon cases are shown in Table 5-2. Similar results were obtained for other numbers of weapons and are included in Appendix C. In Table 5-2, the *Processors* column contains the number of processors utilized to obtain the corresponding mean processing times ($Time$) reported. The number of processors used is also the number of partitions made on the input cost matrix.

Table 5-2. Timing and Speedups of the Level 1 Implementation

Weapons	Targets	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
96	96	1	8.9020	1.00	4.67
96	96	2	1.4335	6.21	25.10
96	96	4	0.5180	17.19	69.47
96	96	8	0.2206	40.35	163.14
96	96	16	0.1028	86.60	350.08
96	96	32	*0.0494	*180.20	*728.50
96	480	1	7.7080	1.00	—
96	480	2	3.7935	2.03	—
96	480	4	1.8853	4.09	—
96	480	8	0.9464	8.14	—
96	480	16	0.4772	16.15	—
96	480	32	0.2422	31.82	—
96	960	1	15.0570	1.00	—
96	960	2	7.5195	2.00	—
96	960	4	3.7635	4.00	—
96	960	8	1.8891	7.97	—
96	960	16	0.9518	15.82	—
96	960	32	0.4836	31.13	—

The $S_{B\&L}$ speedups shown in Table 5-2 are all superlinear for the 96-weapon, 96-target cases. Some of the $S_{B\&L}$ speedups for the 1:5 ratio cases were slightly better than perfect (perfect speedup = number of processors utilized), while the 1:10 ratio cases (96 × 960) were slightly less than perfect as more processors were utilized. One reason why the speedups became less than perfect as the ratio of weapons-to-targets increased is directly related to how the processing times behave. In the B&L algorithm, when the cost matrix is square (i.e., the number of weapons equals the number of targets), the initial solution calculated is nearly always not optimal and must be reshuffled to obtain the optimal solution. However, as the

input cost matrix becomes more and more rectangular, the initial solution more often than not *is* optimal and the reshuffling portion of the B&L algorithm not performed. This results in a time savings and the less than linear increase in the B&L algorithm processing times as the number of weapons is held constant and the number of targets is increased.

When the original cost matrix is divided into more and more partitions, the resulting partitions are increasingly more rectangular. This results in faster computations of the partition assignments because the reshuffling portion of the algorithm is bypassed and results in superlinear speedups over the single-node processing time. This is mainly true for the case when the original cost matrix is square. When the original cost matrix is rectangular, then the previously described behavior is already in effect in the single-node processing times. The partitioning still produces more rectangular submatrices, but the relative reduction in processing times is not as great and results in the more expected near-linear speedups shown in Table 5-2.

The speedups over the sorting method S_{Sort} are only shown for the 1:1 ratio case because the level 0 1:5 and 1:10 cases for 96 weapons were not run as previously explained. For the single-processor case, which is equivalent to a sequential B&L algorithm, the level 1 processing times are more than four times faster than the level 0 times. When multiple processors are utilized, the speedup S_{Sort} becomes very large and illustrates the speed advantage of the parallel level 1 implementation.

A regression analysis similar to the one explained in the level 0 presentation was performed using the processing times shown in Table 5-2 and Appendix C. The resulting models of the processing times were of the same order as the complexity estimate made for the B&L algorithm in Chapter 4. The models differed from the estimate in the coefficients of the terms and some of the lower order terms were not significant. The coefficients are different because of the previously mentioned relationship between high-level operations and lower-level machine instructions. Also, the estimate was based on an assumed worst-case scenario, while the data used in

these performance runs was not worst-case. As an example, the model obtained for the eight-processor, 1:1 weapon-to-target ratio is given in the following equation:

$$processing\ \ time = -5.6283 \times 10^{-8}(WT^2) + 0.00002964(T^2) \qquad (5-3)$$

The adjusted R^2 coefficient for this model was 0.9924 and the probability of rejection was less than 0.0001. Similar models were obtained for other numbers of processors and weapon-to-target ratios. One concern is the negative sign of the leading term. This indicates that the T^2 and WT^2 terms are interactive and to some extent cancel each other out. Each term was modeled individually and yielded acceptable models. The best fit was obtained, however, when both terms were combined into a single model. The Pearson correlation coefficient between the predicted number of operations and the actual processing times was very significant, which indicates that there is some relationship between the two. For example, the Pearson coefficient between the estimated number of operations and the 1:1 weapon-to-target ratio processing times was 0.98397 and the probability of rejection was 0.0160.

The assignment costs and other information for the 96-weapon case are shown in Table 5-3. The column labeled *% Effective* is defined the same as in level 0. An additional column named *% Wasted* contains the percentage of weapons that were redundantly paired with a previously assigned target. These weapons were therefore "wasted" on a target that was already "killed" by another weapon.

One trend that should be noted in Table 5-3 is that as the ratio of weapons-to-targets increases, the % Effective also increases for the same number of processors. This is a result of fewer redundant weapon allocations, which are in turn a result of the larger number of possible targets. The assignment costs are also lower for the higher ratios of weapons-to-targets due to the wider choice of possible targets which may be engaged by each weapon. In all cases, the % Effective drops as more processors are used because none of the processors coordinate the assignments made

Table 5-3. Assignment Results of the Level 1 Implementation

Weapons	Targets	Processors	Cost	% Effective	% Wasted
96	96	1	1779.2	100.0	0.0
96	96	2	1648.0	72.3	27.7
96	96	4	1576.0	65.8	34.2
96	96	8	1544.0	62.7	37.3
96	96	16	1510.4	61.0	39.0
96	96	32	1491.2	60.6	39.4
96	480	1	756.8	100.0	0.0
96	480	2	756.8	89.6	10.4
96	480	4	755.2	84.8	15.2
96	480	8	755.2	83.1	16.9
96	480	16	752.0	83.6	16.4
96	480	32	755.2	82.1	17.9
96	960	1	723.2	100.0	0.0
96	960	2	723.2	92.1	7.9
96	960	4	723.2	87.7	12.3
96	960	8	723.2	84.6	15.4
96	960	16	723.2	83.3	16.7
96	960	32	723.2	82.5	17.5

within each partition. In later implementations, different levels of coordination are introduced in an attempt to reduce the redundant weapon allocations and increase the % effective utilization.

5.2.3 Level 2 The level 2 implementation introduces a small amount of coordination between groups of processors in order to reduce the number of redundant assignments. The 96-weapon timing and speedup results are shown in Table 5-4. The $S_{B\&L}$ and S_{Sort} speedups shown in Table 5-4 were calculated in the same manner described in the level 1 presentation. The *Cntrl* column refers to the number of partitions or controller groups used in the configuration. The *Proc/Cntrl* column refers to the number of processors per controller. The *Tot Proc* column contains the total number of processors utilized in a particular configuration and is derived by multiplying the number of controllers by the number of processors per controller and then adding the number of controllers to the product. For example, a two controller configuration with four processors per controller will utilize $(2 \times 4) + 2 = 10$ processors.

Table 5-4. Timing and Speedups of the Level 2 Implementation

Weapons	Targets	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
96	96	2	2	6	0.659	13.51	54.61
96	96	2	4	10	0.375	23.74	95.97
96	96	2	8	18	0.239	37.25	150.58
96	96	4	2	12	0.189	47.10	190.41
96	96	4	4	20	0.149	59.74	241.53
96	96	8	2	24	*0.046	*193.52	*782.35
96	480	2	2	6	1.983	3.89	—
96	480	2	4	10	1.044	7.38	—
96	480	2	8	18	0.606	12.72	—
96	480	4	2	12	0.951	8.11	—
96	480	4	4	20	0.523	14.74	—
96	480	8	2	24	0.468	16.47	—
96	960	2	2	6	4.615	3.26	—
96	960	2	4	10	2.359	6.38	—
96	960	2	8	18	1.228	12.26	—
96	960	4	2	12	2.305	6.53	—
96	960	4	4	20	1.176	12.80	—
96	960	8	2	24	1.163	12.95	—

The processing times and speedups shown in Table 5-4 are divided into sections corresponding to the 1:1, 1:5, and 1:10 weapon-to-target ratios. Each of these sections can be further subdivided into three sub-sections by the number of controller groups (Cntrl). By grouping in this manner, the effects of adding additional processors per controller can be seen. The processing times for the 1:10 and 1:5 weapon-to-target ratios decreased in proportion to the number of additional processors per controller group: doubling the processors per controller reduced the processing times by approximately half. In the 1:1 ratio case, the reduction in processing times was not as evident. This is related to the processing time behavior discussed in the level 1 presentation. The 1:5 and 1:10 ratio cases provide more choices for allocating weapons and the solution is obtained quicker due to the highly rectangular partitions. Although the partitions in the 1:1 case are also rectangular, there are fewer targets to choose from and computing the partition solutions is more likely to require iterations of the reshuffling portion of the B&L algorithm.

As in the level 1 implementation results, the 1:1 ratio cases produced superlin-

ear speedups. However, the $S_{B\&L}$ values were not as large as those of level 1. This is due to the increased processing times which are a result of the additional overhead involved with the coordination within controller groups. The nodes running the controller processes and the assign processes are basically synchronous. When the controller process is active, then the assign processes are idle and vice versa. This creates a situation where there are always idle node processors and reduces the speedups obtainable. The situation for the 1:5 and 1:10 ratios is similar, but the time required by the controller process is longer because of the factor of 5 or 10 increase in the number of targets that must be coordinated. This causes the assign processes to remain idle longer and results in speedups being less than those of the 1:1 cases.

Although the coordination of redundant pairings does comprise a portion of the processing time, the regression models obtained for the level 2 times were very similar to the level 1 models. This indicates that the coordination does not completely dominate the processing time. An example of the type model obtained is shown in Equation 5-4 for the 18-processor, 1:10 weapon-to-target ratio case.

$$processing\ \ time = -1.95412 \times 10^{-9}(WT^2) + 2.9718 \times 10^{-6}(T^2) - 0.263511 \quad (5-4)$$

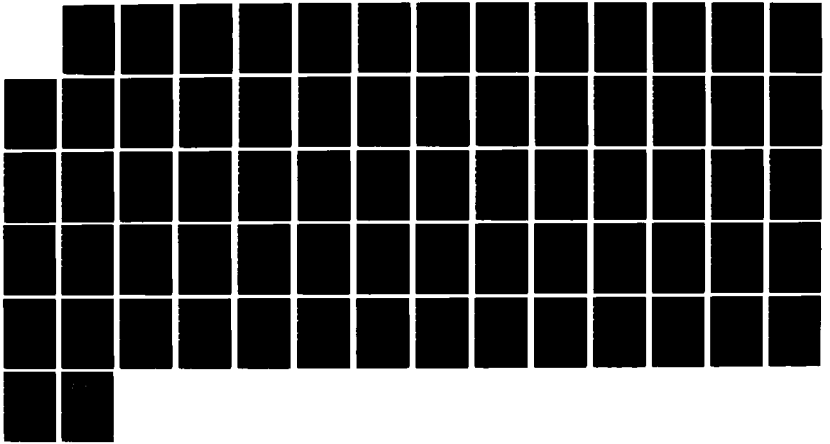
The terms W and T refer to the number of weapons and targets, respectively. The Y-intercept terms were found to be significant in models for this implementation, which is where some of the added computations estimated for the level 2 model are accounted for. The R^2 coefficient was at least 0.95 for all models, which indicates a very good fit between the actual and predicted processing times.

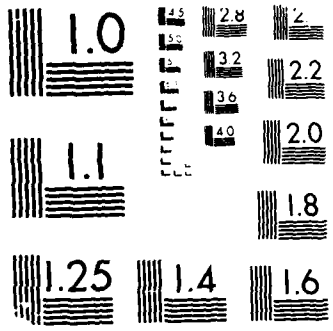
The mean assignment costs, percent weapons effective, percent weapons [illegible] and an additional measure labeled *% Idle* are shown in Table 5-5. The [illegible] sure is unique to this implementation. It is a result of the coordination [illegible] instead of different processes possibly assigning multiple weapons

AD-A198 678 IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI B A CARPENTER

2/2

UNCLASSIFIED DEC 87 AFIT/GCE/ENG/87-2 F/G 12/7 NL





get, the weapons associated with higher cost redundant assignments are placed in an idle state for future use rather than being "wasted" on an already-assigned target. In general, this implementation idled a higher percentage of the weapons when the weapon-to-target ratio was 1:1. This results from the coordination action where, instead of wasting redundantly allocated weapons, they are idled for future use. More redundancies occur in the 1:1 case, which in turn yields a higher percentage of the weapons in an idle state. Some weapons are still wasted because there is no coordination of assignments between controller groups. For the 1:10 ratio, very few weapons were wasted and less than 20% were idled. This stems from fewer redundancies both within and among the controller groups. The greatest advantage of this implementation is that the idled weapons are available for future assignments where they can possibly be used in a more cost effective manner.

Table 5-5. Assignment Results of the Level 2 Implementation

Wpns	Tgts	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Idle	% Wasted
96	96	2	2	6	1328.0	65.8	11.9	22.3
96	96	2	4	10	1180.8	62.7	17.7	19.6
96	96	2	8	18	1115.2	61.0	20.4	18.5
96	96	4	2	12	1372.8	62.7	8.1	29.0
96	96	4	4	20	1281.6	61.0	11.5	27.5
96	96	8	2	24	1436.8	61.0	4.0	35.0
96	480	2	2	6	712.0	84.8	5.4	9.8
96	480	2	4	10	696.0	83.1	7.5	9.4
96	480	2	8	18	689.6	82.7	8.3	9.0
96	480	4	2	12	736.0	83.1	2.5	14.4
96	480	4	4	20	729.6	82.7	1.3	16.0
96	480	8	2	24	748.8	82.7	0.8	16.5
96	960	2	2	6	686.4	87.7	4.8	7.5
96	960	2	4	10	662.4	84.6	7.9	7.5
96	960	2	8	18	651.2	83.3	9.4	7.3
96	960	4	2	12	699.2	84.6	3.1	12.3
96	960	4	4	20	688.0	83.3	4.6	12.1
96	960	8	2	24	712.0	83.3	1.5	15.2

5.2.4 Level 3 The level 3 implementation introduces more coordination between the same configuration of processors found in the level 2 program. After each iteration of the B&L algorithm in the "assign" processors, the controller eliminates

the redundant assignments and sends out vectors containing information to be used in the computation of new assignments.

Table 5-6. Timing and Speedups of the Level 3 Implementation

Wpns	Tgts	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
96	96	2	2	6	0.9730	9.15	36.99
96	96	2	4	10	1.8905	4.71	19.04
96	96	2	8	18	3.5370	2.52	10.17
96	96	4	2	12	2.1135	4.21	17.03
96	96	4	4	20	2.5548	3.48	14.09
96	96	8	2	24	1.4069	5.48	25.58
96	480	2	2	6	2.4975	3.09	—
96	480	2	4	10	4.3405	1.78	—
96	480	2	8	18	6.0985	1.26	—
96	480	4	2	12	4.0150	1.92	—
96	480	4	4	20	4.6833	1.65	—
96	480	8	2	24	2.6573	2.90	—
96	960	2	2	6	4.8910	3.08	—
96	960	2	4	10	8.4135	1.79	—
96	960	2	8	18	11.6000	1.30	—
96	960	4	2	12	7.6133	1.98	—
96	960	4	4	20	9.0953	1.66	—
96	960	8	2	24	5.2775	2.85	—

This implementation was very expensive in terms of processing times as illustrated by the times and speedups in Table 5-6. Except for the first entry in Table 5-6, none of the $S_{B\&L}$ speedups were better than perfect. The extra iterations of the B&L algorithm, combined with the coordination process, substantially increased the processing times and thereby reduced the speedups. As more processors were added to a controller partition, the processing time *increased* rather than decreased. The reason the processing times increased so dramatically is that if redundancies remain after an iteration, then new information vectors must be assembled and *all* processors must recompute another assignment on their given partition based on the updated information. This procedure continues until all redundancies are eliminated. The increase in processing times is an especially undesirable effect since the objective is to decrease rather than increase the time as more processors are utilized.

Regression analyses yielded very similar models for this implementation when

compared to the level 1 and level 2 models. One difference in the models obtained for this implementation is that the Y-intercept term became more significant and positive. This is an indication that there are increasing overheads involved with the coordination process. The estimation of the extra computation involved with the "controller" process made in Chapter 4 could not be confirmed because the regression models were computed with the number of processors held constant. The SAS software regarded the models with processors as a variable as "not of full rank." This means the results obtained would be misleading and biased because of certain inter-relationships between the different terms in the model. The model shown in Equation 5-5 is for the 1:1 ratio, 32-weapon, 20-processor case.

$$processing\ \ time = -5.924 \times 10^{-7}(WT^2) + 0.0000487T^2 + 0.267500 \qquad (5-5)$$

Assignment results for the level 3 program are shown in Table 5-7. The percent weapons wasted decreased as the ratio of weapons-to-targets was increased. This trend was also noted on the results for other numbers of weapons. The assignment costs for level 3 were higher than any of the other implementations. This can be at least partially explained by the method used to reassign weapons that were redundantly allocated. In cases when another target must be selected because of a redundancy, it will be at least equal to and probably a higher cost than the originally selected target. The combination of several substantially higher cost reassignments drives up the average cost dramatically as shown by the cost data in Table 5-7.

5.2.5 Level 4 The level 4 implementation is an attempt to perform several of the tasks of the sequential B&L algorithm in parallel. As the timing and speedup results in Table 5-8 illustrate, the effort did not perform as well as one would hope. The major bottleneck was the amount of interprocessor communications required to update global information used in the selection of potential weapon-to-target pairings.

Table 5-7. Assignment Results of the Level 3 Implementation

Weapons	Targets	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Wasted
96	96	2	2	6	15696.0	71.3	28.7
96	96	2	4	10	18886.0	68.5	31.5
96	96	2	8	18	17457.6	68.5	31.5
96	96	4	2	12	13412.0	64.9	35.1
96	96	4	4	20	16022.4	62.9	37.1
96	96	8	2	24	7556.8	61.3	38.7
96	480	2	2	6	9344.0	87.3	12.7
96	480	2	4	10	8501.2	86.7	13.3
96	480	2	8	18	10070.4	86.3	13.7
96	480	4	2	12	4572.8	83.5	16.5
96	480	4	4	20	6408.0	83.5	16.5
96	480	8	2	24	2606.6	82.7	17.3
96	960	2	2	6	6779.2	89.8	10.2
96	960	2	4	10	11288.0	88.1	11.9
96	960	2	8	18	11124.8	87.3	12.7
96	960	4	2	12	4798.4	86.3	13.7
96	960	4	4	20	7107.2	85.2	14.8
96	960	8	2	24	2947.2	84.0	16.0

The processing times for this implementation were only marginally better than the results obtained for the sequential B&L algorithm (i.e., level 1, one processor). In some cases, the sequential B&L implementation was actually faster than the level 4 implementation. It is difficult to discern the exact reason for the poor performance. One problem noted during the gathering of the results was that the default number of buffers in the iPSC used to handle internode message traffic was too small. When additional buffers were made available, then the amount of memory remaining was inadequate for processing larger cost matrices. This definitely had an effect on the speed of the level 4 implementation.

Another possible reason for the poor performance is that the parallel algorithm used was too inefficient. One especially time consuming task was the search for independent zero elements. Recall that the cost matrix was partitioned into "windows" that were searched independently and in a certain order by the "parallel" processors. After each parallel processor searched one of its windows, then all of the partial results were transmitted to the serial processor for combination and transmittal to all

Table 5-8. Timing and Speedups of the Level 4 Implementation

Weapons	Targets	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
96	96	2	4.969	1.79	7.24
96	96	4	4.563	1.95	7.89
96	96	8	5.075	1.75	7.09
96	96	16	6.616	1.35	5.44
96	96	32	15.711	0.57	2.29
96	480	2	12.373	0.61	—
96	480	4	6.924	1.11	—
96	480	8	6.851	1.13	—
96	480	16	11.802	0.65	—
96	480	32	32.695	0.24	—
96	960	2	17.522	0.86	—
96	960	4	12.876	1.17	—
96	960	8	11.898	1.27	—
96	960	16	18.001	0.84	—

parallel processors for the next set of window searches. The coordination between processors was necessary in order to insure no redundancies occurred. There may be more efficient methods for performing this and other tasks, but the basic Hungarian method, and the B&L algorithm in particular, may be intrinsically serial and not parallelizable.

The regression model obtained for this implementation was somewhat different from the other models. This was expected because this program was so much different from the other programs. The processing times were affected by the hardware limitations to the extent that multiple runs had to be made with different numbers of system buffers and system memory allocations before the processes would complete normally. The most reliable data obtained was for the cases where four and eight processors were used, so these data were used as the basis for the regression analysis. The model for the 8-processor, 32-weapon, 1:1 ratio case is given in Equation 5-6.

$$processing\ \ time = 0.0000475(WT^2) - 0.00893(T^2) + 0.55860(W) - 8.359 \quad (5-6)$$

A term that was not significant in any of the other models is W. The reason for this is that the vectors transmitted and the combination procedure performed by

the serial processor were all strongly related to the number of weapons.

A feature of the level 4 program is that the weapons are always 100% effective. However, this is not an advantage since the time required to achieve these results is longer than the single-processor level 1 program. The assignment results of the level 4 implementation are shown in Table 5-9. Essentially, the assignment results are identical to those of the level 1 implementation utilizing a single processor.

Table 5-9. Assignment Results of the Level 4 Implementation

Weapons	Targets	Processors	Cost	% Killed
96	96	2	1779.2	100.0
96	96	4	1779.2	100.0
96	96	8	1779.2	100.0
96	96	16	1779.2	100.0
96	480	2	756.8	100.0
96	480	4	756.8	100.0
96	480	8	756.8	100.0
96	480	16	756.8	100.0
96	960	2	723.2	100.0
96	960	4	723.2	100.0
96	960	8	723.2	100.0
96	960	16	723.2	100.0

5.3 Analysis of Results

In this section, the performance results of the different implementations are compared and evaluated. The relative advantages and disadvantages of each implementation are also discussed. Graphs are used to illustrate trends and make additional comparisons between the different implementations.

5.3.1 Computation Times and Speedup The computation times varied widely between the different programs. The fastest times were those of the level 1 and level 2 programs, while the slower times were where those of the level 3 and level 4 programs. This can be mainly attributed to the volume and frequency of communications between processors in the different implementations. As the level of coordination and communications increased, the computation times also increased. The correspond-

ing speedups over the level 0 program and the single-processor level 1 program show the reverse trend since T_N is a divisor in the calculation of speedup. The speedups obtained by the four levels of implementation over the level 0 program for 96×96 cost matrices are illustrated in Figure 5-1.

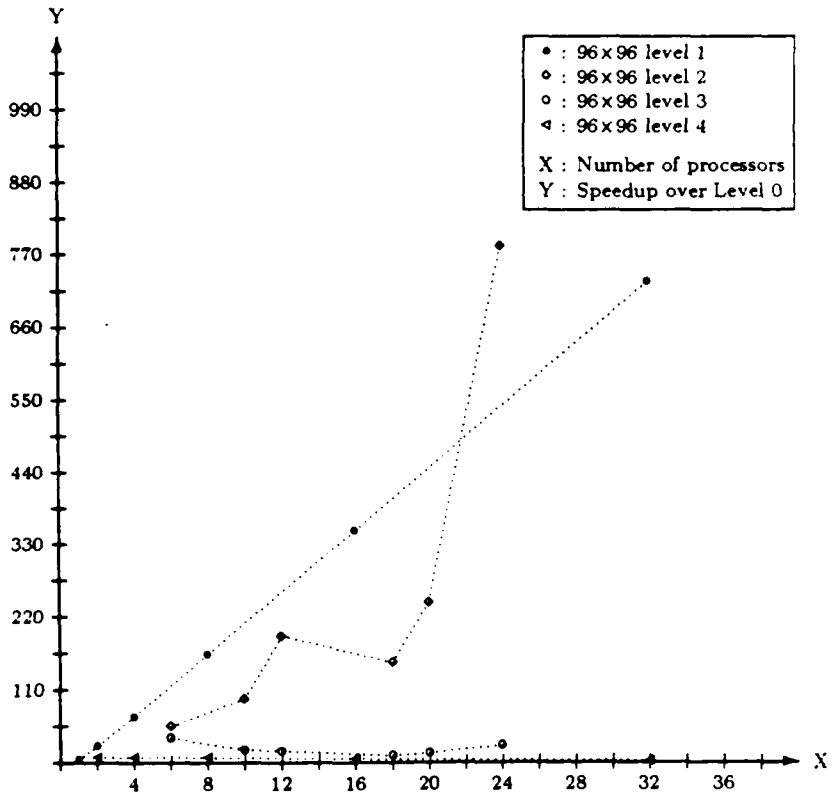


Figure 5-1. Speedups over the Level 0 Implementation (1:1 Weapon-Target Ratio)

The speedups over the single-processor level 1 implementation are shown in Figures 5-2 and 5-3. The single-node level 1 results are equivalent to a sequential version of the B&L algorithm. In all cases, the level 1 and level 2 implementations exhibited substantial speedups over the single-processor programs. The superlinear speedups in the 1:1 ratio cases at first do not seem possible. The explanation for why the B&L algorithm works faster for rectangular matrices than for square matrices was given in the presentation of the level 1 results. Recall from that discussion

that when an initially square cost matrix is partitioned, each processor receives a rectangular submatrix. For rectangular matrices, *it* is more likely that the final assignment solution will be reached in the formation of the initial solution because of the increased number of targets from which assignments can be made. This results in bypassing of some of the reshuffling portions of the B&L algorithm and yields a faster solution to each partition.

In the 1:5 and 1:10 weapon-to-target ratios, the speedups were not as great because the previously discussed performance for rectangular matrices was already in effect for the single-processor times. However, significant speedups were still obtained. One drawback to all of these faster partition solutions is that when they are combined into a final solution, they are no longer optimal because of redundancies, weapons idled, and reassignments to other targets performed by the different implementations. However in most cases, the advantage in processing time allows many sub-optimal assignments to be computed in the time required to compute only one optimal solution.

The speedups obtained by the level 3 and level 4 programs were disappointing. They point out how extensive communications between processors and multiple iterations of the B&L algorithm severely affected the processing times. However, even with the poorer performance when compared with the other parallel implementations, the level 3 and level 4 programs did produce speedups over the sorting method used as the baseline for comparison.

5.3.2 Interprocessor Communications The effects of increased interprocessor communications are illustrated by the longer processing times of the level 3 and level 4 implementations. The ratio of computations-to-communications becomes very small as the level of communications is increased because much more time is spent communicating than computing. The difference in processing times between the level 1 and level 2 programs is not excessive because the coordination process

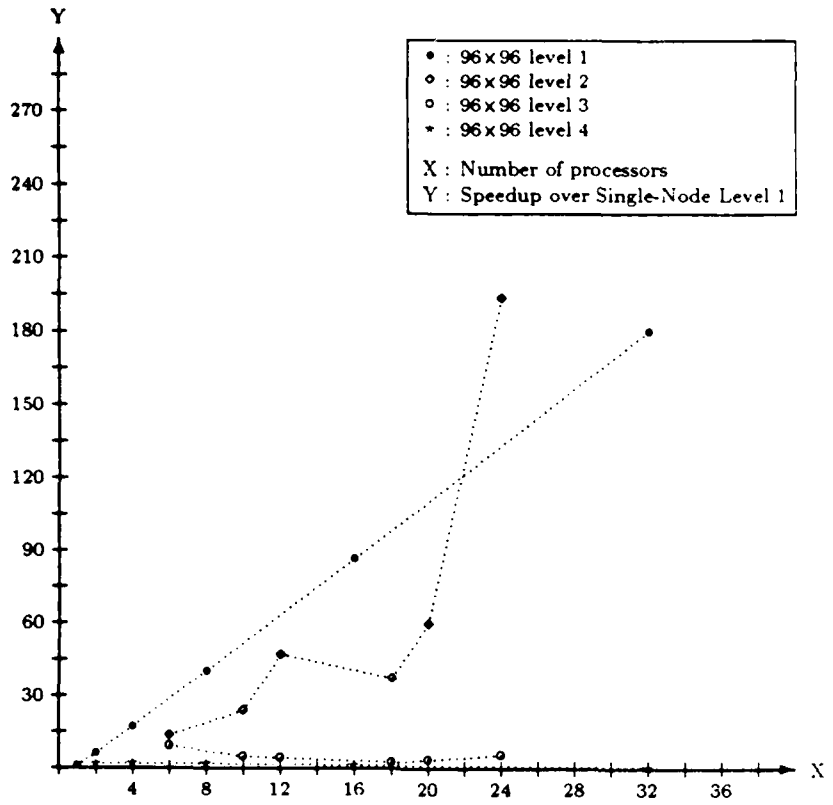


Figure 5-2. Speedups over the Single-Node Level 1 Implementation (1:1 Ratio)

requires a relatively small amount of time. All other operations in the level 1 and level 2 programs are essentially the same. In the level 3 program, the coordination process involves higher interprocessor message traffic to control the extra iterations required to eliminate the redundancies. The transmission of vectors designating the new assignment instructions in level 3 is similar to the vectors transmitted in level 4 for coordinating the search of the matrix partitions for assignments. The processing times of the level 3 and level 4 implementations reflect the extra time spent communicating instead of computing.

The desirability of complete independence between processors can been seen upon comparing the processing times of the level 1 and level 4 implementations. The level 4 implementation is generally less than twice as fast the single-processor

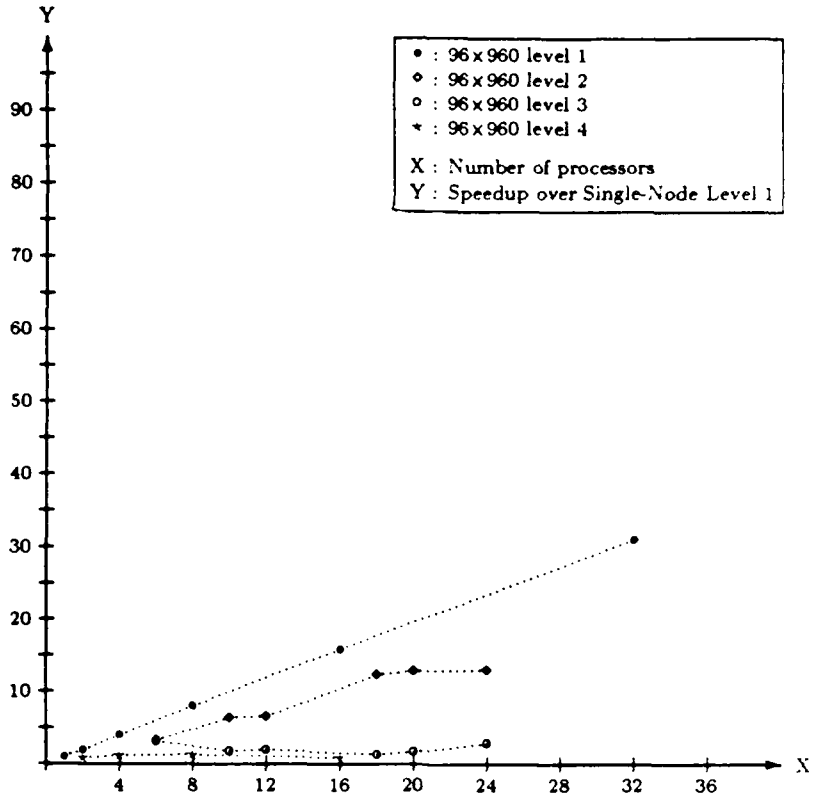


Figure 5-3. Speedups over the Single-Node Level 1 Implementation (1:10 Ratio)

level 1 program. For a 96 weapon, 960 target problem, the level 4 implementation using 8 processors solved the problem only slightly faster than the single-processor level 1 implementation. However, the level 1, 8-processor configuration solved the same problem *6 times faster* than level 4 using 8 processors. However, the resulting assignments and costs were somewhat different: 84.6% of the weapons were effective for level 1 vs. 100% effective for level 4. Except for the 100% effectiveness of the allocated weapons for level 4, the added communications and iterations of the level 3 and level 4 implementations do not appear to provide any particular advantages.

5.3.3 Problem Scalability The relationship between the size of the problem and the size of the machine (number of processors used) is difficult to assess. In the cases studied in this research, the optimum number of processors varied from one

implementation to another. For the level 1 implementation, the 1:1 ratio speedups obtained tended to decrease as the number of weapons and targets increased. For example, the speedup $S_{B\&L}$ for the 32-weapon, 32-target problem using 4 processors was 24.38 while the 128-weapon, 128-target problem speedup using 4 processors was only 18.07. The situation for the 1:5 and 1:10 ratios was completely different. Increasing the number of weapons and targets while holding the weapon-to-target ratio constant provided some interesting results: as the weapons and targets increased, the speedups remained nearly constant. For example, the speedup $S_{B\&L}$ for the 32×320 cost matrix using 16 processors was 15.62. But the speedup for the 128×1280 cost matrix for 16 processors was 15.92. These trends indicate that for weapon-to-target ratios greater than or equal to 1:5, close to perfect speedups are possible even as the problem is greatly enlarged. Some limit to the problem size probably exists, but increasing the problem size by a factor of 16 and still obtaining roughly the same speedup is a good indicator that much larger problems can be solved with reasonably good speedups over the sequential processing times.

Although the $S_{B\&L}$ speedups for the level 2 implementation were not as close to linear as the level 1 results, there were similar trends in scalability. For the 1:5 and 1:10 ratios, the speedups remained fairly constant with a few showing some slight increase as the problem size increased from 32 to 128 weapons. One difference was in the 1:1 ratio cases where, instead of the speedups decreasing as they did in the level 1 implementation, the speedups $S_{B\&L}$ and S_{Sort} also increased slightly as the problem size increased. Based on these results, the level 2 implementation also appears to be a good candidate for solving larger problem sizes.

For both level 1 and level 2 weapon-to-target ratios greater than 1:1, an increase in the weapon-to-target ratio appears to decrease the speedups obtained. By observing the plots in Figure 5-4, these speedups appear to be close to linear. But there is a slight difference between the 1:5 and 1:10 plots. For the level 1 implementation, if there were a 128 processor machine available, the speedups would approach

128 and so on for larger machines. For the level 2 implementation, the speedups would not be as great as the level 1 speedups, but as the size of the machine increased, there would be a corresponding improvement in the processing times and speedups obtained. These are conjectures, but they are based on observations and trends of the data collected. There is no way to predict precisely what the behavior of the processing times would be for larger machines. However, for the range of problem and machine sizes tested, it is reasonable to expect similar results for larger machines and problems.

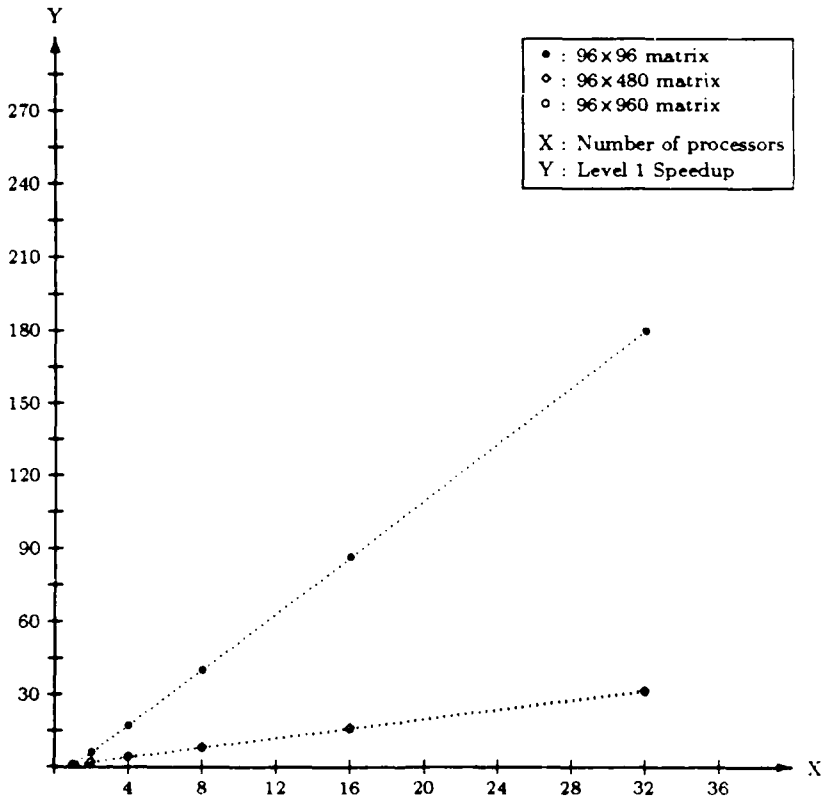


Figure 5-4. Level 1 Speedups over the Sequential B&L Algorithm (96 Wpns)

Up to this point, the discussion has focused on the level 1 and level 2 implementations because the programs are very similar and the speedups obtained were the largest. For the other implementations, the speedups rapidly fell victim to the

communications overhead as the number of processors were increased. Except for the level 1 and level 2 implementations, the processing times generally increased rather than decreased as more processors were used. Because of the trends in processing times observed in the level 3 and level 4 implementations, the extension to larger problem sizes and correspondingly larger numbers of processors does not seem to be feasible.

5.3.4 Cost and Effectiveness of Assignments The processing times and speedups have been the main measures of performance emphasized until now. The manner in which the available weapons are utilized is also very important. If an algorithm is extremely fast but yields poor weapons utilization, it will not be very useful. The assignment results of all the implementations were presented in the previous section. For comparison, plots of weapon effectiveness for 1:1 and 1:10 weapon-to-target ratios are shown in Figures 5-5 and 5-6.

One important trend to note between Figures 5-5 and 5-6 is that the percentage of targets killed increases as the ratio goes from the less likely 1:1 (96×96) ratio case to the more likely 1:10 (96×960) ratio. All of the implementations produced kill percentages above 80% for the 1:10 and 1:5 weapon-to-target ratios. Except for the 100% kill percentages for the relatively slow level 0 and level 4 implementations, the best overall assignment performance was obtained with the level 2 program. In most all 1:5 and 1:10 ratio cases, it wasted less than 10% of the weapons. In general, the level 2 program idled more weapons than it wasted while yielding kill percentages comparable to the other implementations. The idling of weapons rather than wasting them is important, especially when weapons are scarce. Idled weapons can be withheld until a later assignment iteration when they may be utilized in a more cost effective manner.

The associated assignment costs are shown in Figures 5-7 and 5-8. The assignment cost is a measure of how expensive the overall assignment will be in terms of

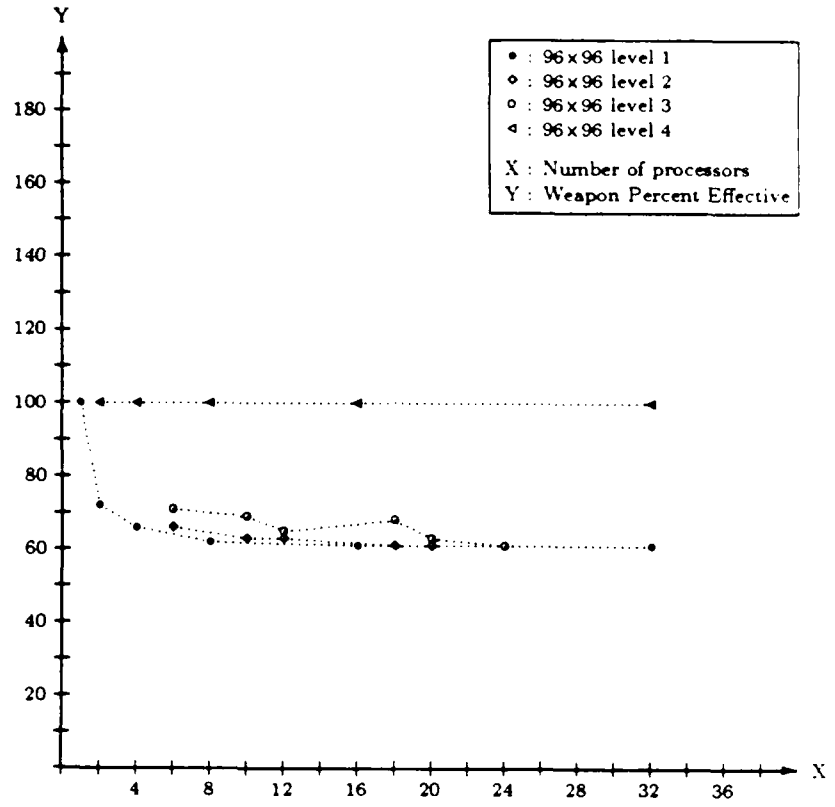


Figure 5-5. Weapon Effectiveness vs. Number of Processors (1:1 Ratio)

resources utilized. The highest costs were produced by the level 3 program. This was explained earlier as a result of the assignment of certain weapons to higher cost targets when redundancies occur. The purpose of the level 3 program was to utilize as many of the weapons as possible to kill all possible targets. Situations may occur when this strategy may be useful. However, upon comparing the results of other programs, level 2 killed approximately the same percentage of targets at a generally lower cost and wasted fewer weapons. In addition, the level 2 processing times were much faster than the level 3 program.

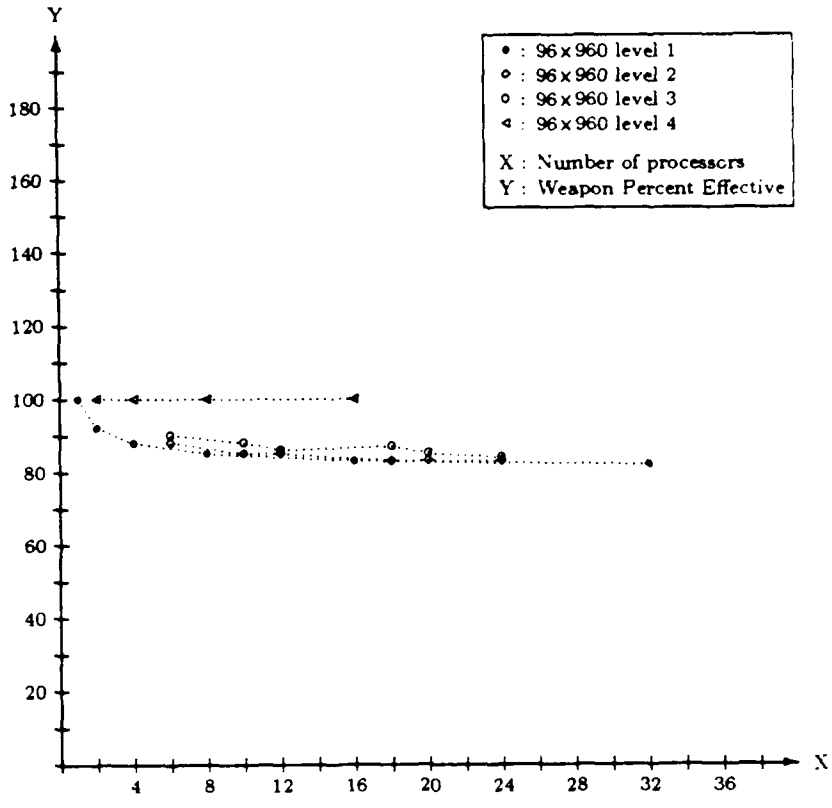


Figure 5-6. Weapon Effectiveness vs. Number of Processors (1:10 Ratio)

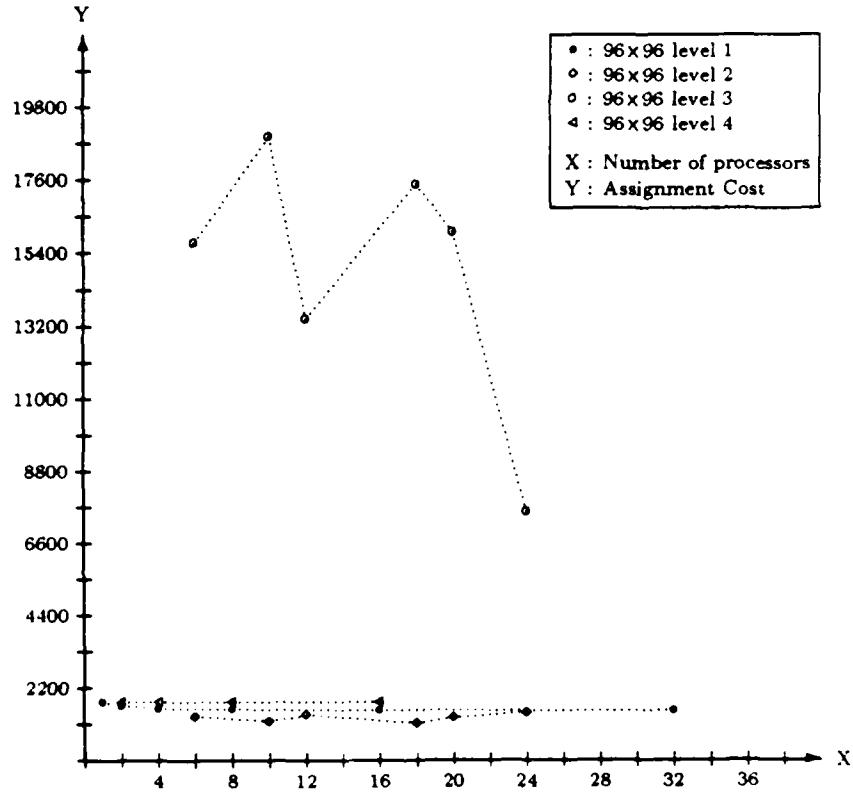


Figure 5-7. Assignment Cost vs. Number of Processors (1:1 Ratio)

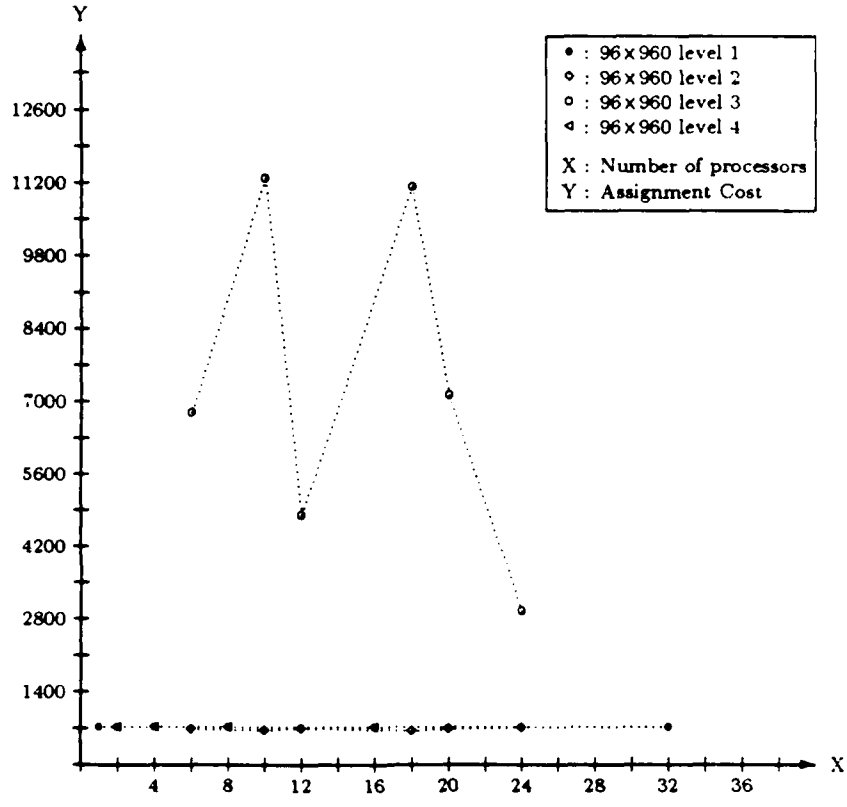


Figure 5-8. Assignment Cost vs. Number of Processors (1:10 Ratio)

5.4 Summary

This chapter has presented and analyzed the performance results of all the programs developed in this research. It first explained the testing approach and then defined the criteria used to measure the performance of the implementations. The results for each program were presented and followed by an assessment of the performance characteristics. Regression analyses provided some insight into how the processing times behaved with the addition of coordination and communications. The speedups and processing times of all implementations were compared and analyzed. Also, the communications overhead, scalability, and effectiveness of the assignments were evaluated. The level 2 program, which involved a modest amount of coordination and communications, produced the best overall performance.

6. *Conclusions and Recommendations*

Before the conclusions and recommendations of this research are presented, a review of the research is in order. Beginning in Chapter 1, an overview of parallel processing as it relates to SDI was presented. The general problem of assignment was introduced and its importance to the BM/C3 system emphasized. The objectives and assumptions were stated in order to define a reasonable scope to the research. Chapter 2 presented a detailed background on parallel processing encompassing the architectures of parallel processors, the hardware organization of the Intel hypercube computer, the techniques for developing parallel software implementations, and a survey of recent parallel implementations developed in the field.

Chapter 3 defined the assignment problem and reviewed some of the important sequential algorithms developed to solve the assignment problem. The transportation and the Hungarian algorithms were chosen for comparison and evaluation. The Hungarian method was chosen as the basis for the parallel implementations. The divide and conquer strategy was chosen as the high-level parallel strategy for combining the partial problem solutions into an overall solution. In Chapter 4, the implementations of two sequential and four parallel assignment programs were explained. The complexity of each implementation was estimated based on high-level operations. Three of the parallel programs utilized the sequential B&A algorithm and involved different types of partitioning and interprocessor communications. The fourth parallel implementation was a parallelized version of the B&A algorithm. Chapter 5 presented the experimental results and a performance analysis of each of the implementations. The performance measures of computation time, speedup, load-balancing, and problem scalability were evaluated.

The remainder of this chapter will focus on the implications of this research and form some conclusions. It will end with recommendations for applications of

this research and topics for further research in the area of parallel processing and BM/C3.

6.1 Parallel Processing: Lessons Learned

The four parallel implementations completed in this research all served to illustrate certain advantages and disadvantages of parallel processing. The first and foremost disadvantage is that all problems cannot be solved in parallel. In some cases, the computational overheads and interprocessor communications overpower any advantage gained by performing certain operations in parallel. This was illustrated by the poor performance of the level 4 implementation where several operations were attempted in parallel. The main problem with the level 4 implementation was the method used to decompose the problem. The "windows" were used to allow multiple processors to search for possible assignments and insure that none of those assignments were redundant. There are other methods for storing portions of matrices in different processors where the data is more easily accessible. But an underlying problem with the B&A algorithm in particular and the Hungarian method in general is that a large number of its operations appear to be intrinsically serial in nature. In the final analysis, the time penalty for parallelizing the operations of the B&A algorithm was just too great. Much better performance was achieved with the level 2 implementation where minimal amounts of communications were used. In the level 2 implementation, a sequential algorithm was used to solve partitions of the overall problem in parallel. A small amount of communications also proved to be better than no communications at all. This was illustrated by the improved assignment results and minimal time penalty of the level 2 program over the non-communicating level 1 program.

The size of the problem partitions also play an important role in how well a parallel implementation performs. In this research, problem solutions utilizing a larger number of small partitions produced noticeably better results than did a

small number of larger partitions. One reason for this appears to be a function of the sequential algorithm used in the node processors. Other algorithms may or may not yield the same results.

Another problem observed is that the balancing of the computational load between the processors has an important effect on the performance. The load balance of the level 1 and level 2 programs appeared to be relatively even. The problem arose in the level 3 and level 4 programs. The controller processor in level 3 became the bottleneck to completing the problem solution. After the assign processors completed one iteration of the assignment algorithm, they remained idle until the controller processor completed a serial process to determine if further processing was needed. While the assign processors computed another iteration, the controller processors remained idle.

In summary, achieving fast and efficient parallel processing appears to rely on three fundamental rules: (1) The problem must be partitionable into a number of independent subproblems. (2) The communications between the processing elements must be kept to a minimum. (3) The computations performed by each processor must be approximately equal and simultaneous.

6.2 Areas of Application

The assignment problem solved in this research was very general. In Chapter 3, the background information on assignment algorithms revealed that many types of problems can be solved using the same basic techniques. Areas such as circuit board routing, network flow analysis, and allocation of resources were cited. The application of weapon-target assignment algorithms is certainly not limited to the missile defense system proposed by the SDIO. Smaller-scale battle management systems could also benefit from the application of parallel assignment algorithms to aid in speeding up the decision processes. In addition, other functions of battle management where fast processing of large amounts of data is necessary could certainly be

performed in parallel. The implementations could be realized using the development techniques and guidelines presented in this research.

6.3 Recommendations for Further Research

The results of this research show that significant decreases in processing times are possible by using multiple processors. The performance of the level 2 implementation illustrated that there needs to be a balance between communications and computations. The Intel iPSC used in this research is a loosely-coupled parallel processor machine. A shared-memory machine described in Chapter 2 was not available for use when this research began, but one has recently been obtained by the department. A natural extension would be to compare the results obtained in this research with the results of assignment algorithms implemented on the shared-memory machine. The reduction in interprocessor message-passing and the sharing of assignment information between processors through the common memory could prove interesting.

Different types of heuristics for reducing the redundant assignments could also be a topic for further research. The elements of the cost matrix were random values in a specified range. Time did not permit experimentation with the effects of different groups of weapons that have similar opportunities for engaging the same targets. The method of deriving the cost information could also be expanded and improved. Instead of random numbers for cost values, further work with simulation programs could be done to derive more representative values. The inclusion of statistical probabilities of target kills based on specific weapons and targets is another possibility.

In Chapter 4, the assignment process was assumed to be memoryless. The assignment depended only on the present state of the system and the present input. Expansion of the implementations to include consideration of past assignments could yield improved results. In addition, methods to predict possible trends in fu-

ture assignments could also be beneficial, especially in situations where the weapon resources are expected to be limited or very expensive.

In closing, this research has demonstrated that parallel processing provides benefits and creates liabilities. Some of the benefits were demonstrated in the computation times and speedups obtained with the implementations. But the results were not completely optimal. This is, of course, just one of the liabilities. Each application will possibly involve tradeoffs of one type or another. Further research in the area of parallel algorithms and parallel software implementations can build on the results presented in this thesis and yield further performance improvements.

Appendix A. *The Transportation Method*

In Chapter 3, the transportation method of solving the assignment problem was briefly described. This appendix presents the steps of the algorithm in detail and points out the similarities between the transportation method and the simplex method from which it was derived. Following the algorithm presentation, an example problem is given that illustrates how the algorithm operates

There are two phases to the transportation method. The first phase is to formulate the initial basic feasible solution. The second phase checks the initial solution for optimality and incrementally improves upon it until it is optimal. There have been several methods devised to provide the initial solution, but one simple approach known as the "northwest corner rule" will be given here [Chu 57].

1-1 Initialize the table by setting all x_{ij} entries to null (no entry) and all c_{ij} entries to the corresponding cost matrix values.

1-2 Beginning with the cell in the northwest corner of the table, assign the minimum of a_i or b_j, which correspond to the row availability (resources) and column requirements (requesters) respectively, to the x_{ij} variable. For the assignment problem, these elements will always be one, so no decision needs to be made. Both a_i and b_j are reduced to zero and the x_{ij} element is set to one.

1-3 Eliminate from further consideration the $i\ th$ row and $j\ th$ column containing the x_{ij} element just modified. This effectively reduces the dimension of the table. If no rows and columns remain after this elimination, the initial solution is complete. Otherwise, repeat Steps 1-2 and 1-3.

When the initial solution is complete, the number of cells assigned will be n and they will form a northwest to southeast diagonal in the matrix. However, one restriction of the transportation method is that the number of assigned cells in an $n \times m$ cost matrix must equal $n + m - 1$. When the number of assigned cells

is less than $n + m - 1$, then the solution is called degenerate. In the assignment problem, the initial solution is always degenerate since only n cells of the required $n + n - 1$ are assigned. Additional *artificial* assignments need to be made to achieve a nondegenerate solution in order for the second phase of the algorithm to work. A method of generating these artificial assignments is described following steps:

1-4 Start with an unassigned cell and assign this cell a $+\theta$ designation.

1-5 A θ-path is a loop that begins and ends on a particular unassigned cell by alternately assigning $+\theta$ and $-\theta$ designations to certain assigned cells in the loop. This θ-path loop is formed by making one or more horizontal *and* vertical movements. Except for the initial and final movements from and to the selected unassigned cell, each movement must be from one assigned cell to another and form a segment with assigned cells as endpoints by traversing one or more cells per movement. One subtlety of forming the θ-path is that all assigned cells do not need to be included in the path and some cells may be "skipped over" when forming the path. If a closed loop can be formed in this fashion, the unassigned cell is termed *dependent*. The objective is to identify all *independent* cells, which are those cells where a closed-loop θ-path cannot be formed. Once all independent unassigned cells are identified, then a sufficient number of *artificial* ϵ allocations are made to these cells with the lowest cost c_{ij} to form the required $n + n - 1$ assignments. An ϵ allocation is defined as a very small positive number which will be set to zero in the final solution to obtain the actual allocation [Ign82].

Once the required number of ϵ allocations are made, then the steps of phase II can be performed as follows:

2-1 Add an additional column and row to the table to contain row indicators R_i and column indicators K_j.

2-2 Given a non-degenerate initial solution from phase I, assign a zero element to any of the R_i or K_j positions.

2-3 For each cell that has an actual or artificial value for the $x_{i,j}$ entry, satisfy the following expression:

$$\Delta_{i,j} = R_i + K_j + c_{i,j} = 0 \qquad (A-1)$$

The initial zero R_i or K_j element can be used to determine the missing R_i or K_j element. From this initial determination, all other values of R_i and K_j can be determined [Ign 82].

2-4 For the remaining unassigned cells, determine the values of $\Delta_{i,j}$ using the corresponding R_i, K_j, and $c_{i,j}$ values. Enter these values of $\Delta_{i,j}$ into the associated cell in the upper right hand corner.

2-5 If all of the $\Delta_{i,j}$ values are nonnegative for the unassigned cells, then the assignment is optimal and the algorithm terminates. If any $\Delta_{i,j}$ values are negative, then the solution can be improved and step 2-6 must be performed.

2-6 Select the unassigned cell with the most negative $\Delta_{i,j}$ value. In the case of a tie in $\Delta_{i,j}$ values, choose one of the most negative $\Delta_{i,j}$ cells arbitrarily. This step is analogous to the simplex method of selecting a non-basic variable to enter into the basic solution set. The present assignment must be changed in order to include this new variable, which requires that one of the present assigned cells (basic variables) be removed. Go to Step 2-7.

2-7 Construct a θ-path as described in step 1-5, beginning with the cell having the most negative $\Delta_{i,j}$ value. However, this time the objective is to form a *closed-loop* θ-path.

2-8 The results of step 2-7 will yield some cells with $+\theta$ designations and others with $-\theta$ designations. The cells with $+\theta$ designations will become assigned cells with $x_{i,j}$ values of one and all $-\theta$ cells become unassigned (no entry for $x_{i,j}$). This step is the same as the simplex method of selecting basic variables that are to leave the basic solution set.

2-9 If the new assignment obtained in Step 2-8 is degenerate, perform steps 1-4 and 1-5 to add the required number of ϵ allocations to form a nondegenerate solution. Then repeat steps 2-2 to 2-8 until an optimal solution is indicated in step 2-5.

As an example of the transportation algorithm just described, consider the following cost matrix:

$$A_0 = \begin{bmatrix} 7 & 4 & 3 & 8 \\ 5 & 5 & 4 & 9 \\ 2 & 7 & 9 & 2 \\ 10 & 3 & 1 & 6 \end{bmatrix} \qquad (A-2)$$

Using the c_{ij} values from this matrix, the initial transportation table can be formed as shown in Table A-1. The null x_{ij} values are indicated by a "–".

Table A-1. Initial Transportation Table

requester → resource ↓	1	2	3	4	a_i ↓
1	– 7	– 4	– 3	– 8	1
2	– 5	– 5	– 4	– 9	1
3	– 2	– 7	– 9	– 2	1
4	– 10	– 3	– 1	– 6	1
b_j →	1	1	1	1	4

The initial basic feasible solution resulting from steps 1-2 and 1-3 of the transportation algorithm is shown in Table A-2. The a_i column and the b_j row will be omitted in later representations since they will not be modified.

Table A-2. Initial Basic Feasible Solution

requester → resource ↓	1	2	3	4	a_i ↓
1	1 7	– 4	– 3	– 8	0
2	– 5	1 5	– 4	– 9	0
3	– 2	– 7	1 9	– 2	0
4	– 10	– 3	– 1	1 6	0
b_j →	0	0	0	0	0

One possible result of performing steps 1-4 and 1-5 is given in Table A-3. All unassigned cells in Table A-2 are independent. Three additional assignments are needed to form the required 4 + 4 - 1 = 7 assignments. The three independent cells with the lowest c_{ij} costs were selected and given the ϵ allocations as shown in Table A-3. One note of explanation about the choice of ϵ assignments is warranted. The assignment of an ϵ to cell x_{42} instead of x_{34} was necessary because after cell x_{43} was assigned, the θ-path for cell x_{34} was no longer independent. This means that the lowest cost unassigned cells are not always given the ϵ allocations.

Now that a nondegenerate basic solution has been obtained, phase II of the transportation method can be entered to determine if this initial solution is optimal. If the assignment is not optimal, then it will be modified to improve it.

As required by Step 2-1, the additional R_i column and K_j row are added to the table. An initial zero element is arbitrarily assigned to the R_4 position by Step 2-2 and is shown in Table A-4. Any of the other positions could have been chosen for this initial zero element.

The first iteration of step 2-3 is shown in Table A-5 where the $\Delta_{ij} = R_i + K_j + c_{ij}$ expression is satisfied for the assigned cells in row 4. There are several iterations

Table A-3. Initial ϵ Assignments

requester → / resource ↓	1	2	3	4
1	1 7	– 4	– 3	– 8
2	– 5	1 5	– 4	– 9
3	ϵ 2	– 7	1 9	– 2
4	– 10	ϵ 3	ϵ 1	1 6

Table A-4. Initial Row Indicator R_i Assignment

requester → / resource ↓	1	2	3	4	R_j ↓
1	1 7	– 4	– 3	– 8	
2	– 5	1 5	– 4	– 9	
3	ϵ 2	– 7	1 9	– 2	
4	– 10	ϵ 3	ϵ 1	1 6	0
K_j →					

required to formulate the remaining R_i and K_j elements. One possible sequence is shown in Tables A-6 and A-7.

Table A-5. Calculation of Additional R_i and K_j Values

requester → resource ↓	1	2	3	4	R_j ↓
1	1 7	– 4	– 3	– 8	
2	– 5	1 5	– 4	– 9	
3	ϵ 2	– 7	1 9	– 2	
4	– 10	ϵ 3	ϵ 1	1 6	0
K_j →		-3	-1	-6	

Table A-6. Additional R_i and K_j Values

requester → resource ↓	1	2	3	4	R_j ↓
1	1 7	– 4	– 3	– 8	
2	– 5	1 5	– 4	– 9	
3	ϵ 2	– 7	1 9	– 2	-8
4	– 10	ϵ 3	ϵ 1	1 6	0
K_j →	6	-3	-1	-6	

Once all the R_i and K_j values are determined, the $\Delta_{i,j}$ values can be calculated for the unassigned cells and entered into the associated cells. Using the values of R_i and K_j from Table A-7, Step 2-4 yields the results shown in Table A-8.

Table A-7. Complete R_i and K_j Values

requester → resource ↓	1	2	3	4	R_j ↓
1	1 7	– 4	– 3	– 8	-13
2	– 5	1 5	– 4	– 9	-2
3	ϵ 2	– 7	1 9	– 2	-8
4	– 10	ϵ 3	ϵ 1	1 6	0
K_j →	6	-3	-1	-6	

Table A-8. Δ_{ij} Values for Unassigned Cells

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7	– -12 4	– -11 3	– -11 8	-13
2	– +9 5	1 5	– +1 4	– +1 9	-2
3	ϵ 2	– -4 7	1 9	– -12 2	-8
4	– +16 10	ϵ 3	ϵ 1	1 6	0
K_j →	6	-3	-1	-6	

From Step 2-5, since all of the Δ_{ij} values are not nonnegative, the assignment is not optimal and can be improved by performing Step 2-6. The most negative Δ_{ij} value in Table A-8 is -12, which is associated with the cells x_{12} and x_{34}. Ties in the negative values may be broken arbitrarily, so x_{34} is chosen. Now the θ-path must be constructed by using Step 2-7 so that the assignments may be shuffled to bring the x_{34} variable into the basic solution. The resulting θ-path is shown in Table A-9.

Table A-9. θ-Path for Exchange of Variables

requester → / resource ↓	1	2	3	4	R_i ↓
1	1 7	- 4 -12	- 3 -11	- 8 -11	-13
2	- 5 +9	1 5	- 4 +1	- 9 +1	-2
3	ϵ 2	- 7 -4	1 9 $-\theta$	- 2 -12 $+\theta$	-8
4	- 10 +16	ϵ 3	ϵ 1 $+\theta$	1 6 $-\theta$	0
K_j →	6	-3	-1	-6	

Now, the x_{ij} values of the cells in the θ-path must be modified to form the new assignment shown in Table A-10.

The new assignment is degenerate since there are only six assignments and seven are required. Performing Steps 1-4 and 1-5 yields a possible nondegenerate basic solution shown in Table A-11.

The presentation of next iteration of phase II will slightly abbreviated, but the intermediate results of each step will be shown. One possible result of performing Steps 2-2 and 2-3 is shown in Table A-12.

Performing Step 2-4 results in following Table A-13. Since all Δ_{ij} values in Table A-14 are not nonnegative, the solution can be improved upon further. The most negative Δ_{ij} value in Table A-13 is -2 (Δ_{21}). A θ-path beginning with this cell

Table A-10. New Assignment from First Iteration

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7	– 4	– 3	– 8	
2	– 5	1 5	– 4	– 9	
3	ϵ 2	– 7	– 9	1 2	
4	– 10	ϵ 3	1 1	– 6	
K_j →					

Table A-11. Second Nondegenerate Basic Solution

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7	– 4	ϵ 3	– 8	
2	– 5	1 5	– 4	– 9	
3	ϵ 2	– 7	– 9	1 2	
4	– 10	ϵ 3	1 1	– 6	
K_j →					

Table A-12. Second Set of R_i and K_j Variables

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7	– 4	ϵ 3	– 8	-2
2	– 5	1 5	– 4	– 9	-2
3	ϵ 2	– 7	– 9	1 2	+3
4	– 10	ϵ 3	1 1	– 6	0
K_j →	-5	-3	-1	-5	

is shown in Table A-14 and traverses the sequence of cells Δ_{21}, Δ_{11}, Δ_{13}, Δ_{43}, Δ_{42}, Δ_{22}, and Δ_{21} to form a closed loop.

Table A-13. Second Set of Δ_{ij} Values

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7	– -1 4	ϵ 3	– +1 8	-2
2	– -2 5	1 5	– +1 4	– +2 9	-2
3	ϵ 2	– +7 7	– +11 9	1 2	+3
4	– +5 10	ϵ 3	1 1	– +1 6	0
K_j →	-5	-3	-1	-5	

Table A-14. Second θ-Path for the Most Negative Δ_{ij}

requester → resource ↓	1	2	3	4	R_i ↓
1	1 7 $-\theta$	– -1 4 –	ϵ 3 $+\theta$	– +1 8	-2
2	– -2 5 $+\theta$	1 5 $-\theta$	– +1 4	– +2 9	-2
3	ϵ 2	– +7 7	– +11 9	1 2	+3
4	– +5 10	ϵ 3 $+\theta$	1 1 $-\theta$	– +1 6	0
K_j →	-5	-3	-1	-5	

The assignment resulting from reassigning the resources from the $-\theta$ cells to the $+\theta$ cells is shown in Table A-15. Since there are only five assignments in this new table, two additional ϵ allocations must be made using Steps 1-4 and 1-5. One possible set of ϵ allocations is shown in Table A-16.

Table A-15. Second Iteration Assignment

requester → resource ↓	1	2	3	4	R_i ↓
1	– 7	– 4	1 3	– 8	
2	1 5	– 5	– 4	– 9	
3	ϵ 2	– 7	– 9	1 2	
4	– 10	1 3	– 1	– 6	
K_j →					

Table A-16. Third Nondegenerate Basic Feasible Solution

requester → resource ↓	1	2	3	4	R_i ↓
1	– 7	– 4	1 3	– 8	
2	1 5	– 5	ϵ 4	– 9	
3	ϵ 2	– 7	– 9	1 2	
4	– 10	1 3	ϵ 1	– 6	
K_j →					

After completing the ϵ allocations, the new R_i and K_j elements can be determined. One possible arrangement is shown in Table A-17.

Table A-17. Third Set of R_i and K_j Variables

requester → resource ↓	1	2	3	4	R_i ↓
1	– 7	– 4	1 3	– 8	-2
2	1 5	– 5	ϵ 4	– 9	-3
3	ϵ 2	– 7	– 9	1 2	0
4	– 10	1 3	ϵ 1	– 6	0
K_j →	-2	-3	-1	-2	

After determining all the R_i and K_j, the Δ_{ij} values for the unassigned cells can be calculated. The results in Table A-18 show that Δ_{12} and Δ_{22} are still negative, which requires another shuffle of the assignment using the θ-path of Steps 2-6 and 2-7. Possible results of performing these steps are given in Table A-19.

Table A-18. Third Set of Δ_{ij} Variables

requester → resource ↓	1	2	3	4	R_i ↓
1	– +3 7	– -1 4	1 3	– +4 8	-2
2	1 5	– -1 5	ϵ 4	– +4 9	-3
3	ϵ 2	– +4 7	– +8 9	1 2	0
4	– +8 10	1 3	ϵ 1	– +4 6	0
K_j →	-2	-3	-1	-2	

Reassigning the resources according to the θ-path constructed in Table A-19 and assigning new ϵ allocations results in the assignment shown in Table A-20.

Table A-19. Third θ-Path for the Most Negative $\Delta_{i,j}$

requester → resource ↓	1	2	3	4	R_i ↓
1	– +3 7	– -1 4 +θ	1 3 −θ	– +4 8	-2
2	1 5	– -1 5	ϵ 4	– +4 9	-3
3	ϵ 2	– +4 7	– +8 9	1 2	0
4	– +8 10	1 3 −θ	ϵ 1 +θ	– +4 6	0
K_j →	-2	-3	-1	-2	

Possible results of a third iteration of Steps 2-2, 2-3, and 2-4 are represented by Table A-21.

Table A-20. Third Iteration Assignment

requester → resource ↓	1	2	3	4	R_i ↓
1	– 7	1 4	ϵ 3	– 8	
2	1 5	– 5	ϵ 4	– 9	
3	ϵ 2	– 7	– 9	1 2	
4	– 10	– 3	1 1	– 6	
K_j →					

Upon inspecting the $\Delta_{i,j}$ values in Table A-21, all are found to be nonnegative. This means an optimal solution has finally been obtained. After setting the ϵ allocations to zero, Table A-22 summarizes the optimal assignments.

Table A-21. Third Iteration R_i, K_j, and Δ_{ij} Values

requester → resource ↓	1	2	3	4	R_i ↓
1	− +3 7	1 4	ϵ 3	− +4 8	+1
2	1 5	− 0 5	ϵ 4	− +4 9	0
3	ϵ 2	− +5 7	− +8 9	1 2	+3
4	− +8 10	− +1 3	1 1	− +4 6	+3
K_j →	-5	-5	-4	-5	

Table A-22. Results of Transportation Method

Resource	*Requester*
1	2
2	1
3	4
4	3

The total cost of the assignment can be obtained by summing the associated c_{ij} values of the assigned cells and yields

$$5 + 4 + 1 + 2 = 12$$

This is an improvement over the initial assignment from Table A-2, which had a value or cost of

$$7 + 5 + 9 + 6 = 27$$

This completes the example of the transportation method of solving the assignment problem.

Appendix B. *The Hungarian Method*

In Chapter 3, the general approach of Hungarian method for solving the assignment problem was described. In this appendix, a detailed explanation of those steps will be presented. Then an example problem will be used to illustrate how the algorithm operates.

In the presentation that follows, reference to the rating matrix refers to the matrix described in Section 3.1.2. The specific steps of Hungarian method are given in the following:

1. Find the minimum element in each row of the rating matrix A_0 and subtract that element from each element of that row. Next, find the smallest element in each column and subtract that element from each element of that column. The resulting matrix will now contain at least one null element in each row and column. This new, modified matrix will simply be referred to as *the matrix* in later references.

2. Locate any row in the matrix that contains only one null element and suitably mark the null element's position. Cross out all other null elements in the column that contains this marked position. Repeat this process until no more rows can be found with only one null element that has not been marked or crossed out. If all rows contain a marked position, then these positions constitute the optimum assignment. The total cost of the assignment can be found by summing the individual costs of the corresponding positions in the original matrix A_0. Otherwise, if all rows do not contain a marked position, then go to step 3.

3. Locate a column in the matrix from the previous steps that contains only one null element. Mark this position and cross out all other null elements in the row that contains this newly marked position. Repeat this process until no more such columns can be found. If every column contains a marked element, then these

marked positions form the optimum assignment and the cost can be calculated as in step 2. Otherwise, go to step 4.

4. Since an optimal solution has not yet been reached, more null elements must be generated. First, the minimum set of lines that contain or cover all of the null elements in the matrix must be constructed. By disregarding the crossed out elements and retaining the marked elements from steps 2 and 3, the following procedure can be used to draw this minimum set of lines:

- **4.1** Mark the rows that do not contain any marked elements.
- **4.2** Mark the columns that have an unmarked null element in a marked row.
- **4.3** Mark the rows that have a marked null element in a marked column.
- **4.4** Repeat steps 4.2 and 4.3 until no more rows or columns can be marked.
- **4.5** Draw lines through all unmarked rows and all *marked* columns.

5. All elements with lines drawn through them are "covered" and those without lines through them are "uncovered." Find the *smallest uncovered element* in the matrix and subtract this element from all *uncovered elements* in the matrix. Then add this smallest element to all *covered* elements that are located at the intersections of the lines drawn in step 4.5 to form a new matrix. If all elements of the matrix are covered, then this indicates the optimum assignment has been reached and exists in the set of null elements in the present matrix.

This step is a result of the König-Egerváry theorem on the minimum set of covering lines [Kre68]. Its objective is to generate additional *independent* zero elements to be covered by lines in later iterations of the algorithm. As defined in Section 3.2.4, independent means that no other zero elements are present in the same row and column. With N available resources, at least N independent zero elements need to be included in the set of zero elements used to make the optimal assignment. By performing Step 5, the cost of adding these additional zero elements to the solution set is minimized. This step reduces all of the *uncovered* elements by

the same minimum uncovered amount, increases the elements covered twice by the same amount, and does not change the elements covered only once. This procedure is very similar to the simplex method's exchange of basic and nonbasic variables explained in Sections 3.2.1, 3.2.2, and 3.3.1.

6. Repeat steps 2 through 5 until the optimum assignment is found.

As an example of Hungarian method just presented, again consider the cost matrix from Appendix A, which is repeated here for convenience:

$$A_0 = \begin{bmatrix} 7 & 4 & 3 & 8 \\ 5 & 5 & 4 & 9 \\ 2 & 7 & 9 & 2 \\ 10 & 3 & 1 & 6 \end{bmatrix} \qquad (B-6)$$

The matrices referred to in the algorithm will be represented as tables in the following presentation. Using the steps of the Hungarian algorithm, the following tables illustrate the procedure. First, the minimum row elements of A_0 are identified and subtracted to yield the following Table B-1. Then the minimum column elements of Table B-1 are identified and subtracted to form Table B-2.

Table B-1. Results of Subtracting Minimum Row Elements

resource → / requester ↓	1	2	3	4
1	4	1	0	5
2	1	1	0	5
3	0	5	7	0
4	9	2	0	5

Table B-2. Results of Subtracting Minimum Column Elements

resource → / requester ↓	1	2	3	4
1	4	0	0	5
2	1	0	0	5
3	0	4	7	0
4	9	1	0	5

In steps 2 through 5, a box will be used to mark the single null elements in the rows or columns and an '×' to cross out null elements. Performing the procedure of Step 2 yields Table B-3.

Table B-3. Independent Null Row Elements

resource → / requester ↓	1	2	3	4
1	4	[0]	×0	5
2	1	×0	×0	5
3	0	4	7	0
4	9	1	[0]	5

Rows 2 and 3 do not contain a boxed element, so the optimum assignment has not been reached. Step 3 must now be performed and one possible result is shown in Table B-4.

One note of explanation is needed about Table B-4. The null element boxed in row 3, column 1 was not the only choice. The null element in row 3, column 4 could have been boxed and the null element in row 3, column 1 crossed out. Recall that the set of null elements contains at least one optimal assignment and possibly more than one.

Checking the columns containing boxed elements in Table B-4 shows that column 4 does not contain a boxed element, so an optimum assignment has not been

Table B-4. Independent Null Row and Column Elements

resource → requester ↓	1	2	3	4
1	4	[0]	ø	5
2	1	ø	ø	5
3	[0]	4	7	ø
4	9	1	[0]	5

reached. This requires the generation of more null elements, so Step 4 must be taken. Only one row does not contain a boxed element, and Step 4.1 yields Table B-5. Checking for columns with null elements in a marked row as required in Step 4.2 results in Table B-6.

Table B-5. Checked Rows Without Boxed Null Elements

resource → requester ↓	1	2	3	4	Row Checks
1	4	[0]	0	5	
2	1	0	0	5	√
3	[0]	4	7	0	
4	9	1	[0]	5	
Column Checks					

Now, using Step 4.3 requires rows 1 and 4 to be marked since they contain a boxed null element in a marked column. The results are shown in Table B-7. Rechecking Steps 4.2 and 4.3, as required by Step 4.4, reveals that no other rows or columns can be marked. Now Step 4.5 can be followed, which calls for lines to be drawn through all *unmarked* rows and all *marked* columns to form the set of covering lines. The covering lines are illustrated by the asterisks at either end of a row or column in the following Table B-8.

Table B-6. Checked Columns with Null Elements in Checked Rows

resource → requester ↓	1	2	3	4	Row Checks
1	4	[0]	0	5	
2	1	0	0	5	√
3	[0]	4	7	0	
4	9	1	[0]	5	
Column Checks		√	√		

Table B-7. Checked Rows with Boxed Null Elements in Checked Columns

resource → requester ↓	1	2	3	4	Row Checks
1	4	[0]	0	5	√
2	1	0	0	5	√
3	[0]	4	7	0	
4	9	1	[0]	5	√
Column Checks		√	√		

Table B-8. A Minimum Set of Covering Lines

* *
* *

resource → requester ↓	1	2	3	4
1	4	0	0	5
2	1	0	0	5
3	0	4	7	0
4	9	1	0	5

* * * *

* *
* *

Step 5 requires that the minimum *uncovered* element be subtracted from each uncovered element and added to the elements that lie at the intersections of the covering lines. From the previous diagram, the minimum uncovered element is 1. Subtracting this from the proper elements results in Table B-9.

Table B-9. New Table From Step 5

resource → / requester ↓	1	2	3	4
1	3	0	0	4
2	0	0	0	4
3	0	5	8	0
4	8	1	0	4

Performing step 2 again yields Table B-10. Now there is a boxed element in each row of Table B-10, so the optimum assignment has been reached. In this case, the resulting assignments are shown in Table B-11. Note that this table is identical to Table A-23 obtained in the transportation method example in Appendix A.

Table B-10. Independent Null Row and Column Elements

resource → / requester ↓	1	2	3	4
1	3	[0]	ø	4
2	[0]	ø	ø	4
3	ø	5	8	[0]
4	8	1	[0]	5

The cost of this assignment shown in Table B-11 can be obtained by summing the corresponding costs in the original rating matrix A_0 which gives

$$5 + 4 + 2 + 1 = 12$$

Table B-11. Results of Hungarian Method

Resource	*Requester*
1	2
2	1
3	4
4	3

The sum of all the minimum elements found in steps 1 and 5 should equal this assignment cost since these minimum elements represent the costs associated with each intermediate assignment. Summing these values yields

$$3 + 4 + 2 + 1 + 1 + 1 = 12$$

The cost results of the Hungarian method are also identical to the transportation example, as expected. The example just presented was a minimization of the assignment cost. It could have been transformed into a maximization of the assignment cost by modifying the original rating matrix as follows:

0.1 Find the maximum element in the rating matrix. Create a new matrix $C_0 = \|c_{ij}\|$ by individually subtracting each element in the cost matrix from the value of the maximum element and store the difference in the corresponding location in C_0:

$$c_{ij} = \underbrace{Max}_{i,j}(a_{ij}) - a_{ij} \qquad (B-7)$$

One other variation would be the case where the number of requesters and resources were not equal. In this case, the rating matrix would not be square as it must be for the original Hungarian algorithm to work. This can be taken care of by adding "dummy" resources or requesters with rating values of zero [Ign82]. This completes the example and discussion of the Hungarian method.

Appendix C. *Additional Results*

This appendix lists all of the remaining results of the implementations described in this report. All entries marked with a * are more than 10% in error due to the accuracy of the timing function of the iPSC.

Table C-1. Timing and Speedups of the Level 1 Implementation (32 Wpns)

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
32	32	1	1.2750	1.00	1.84
32	32	2	0.2210	5.77	10.64
32	32	4	0.0523	24.38	44.97
32	32	8	*0.0245	*52.04	*96.00
32	32	16	*0.0116	*109.91	*202.76
32	32	32	*0.0061	*209.02	*385.57
32	160	1	0.9100	1.00	23.84
32	160	2	0.4320	2.11	50.22
32	160	4	0.2138	4.26	101.46
32	160	8	0.1070	8.50	202.74
32	160	16	0.0551	16.52	393.70
32	160	32	*0.0285	*31.93	*761.16
32	320	1	1.7090	1.00	29.88
32	320	2	0.8400	2.03	60.78
32	320	4	0.4215	4.05	121.13
32	320	8	0.2135	8.00	239.14
32	320	16	0.1094	15.62	466.70
32	320	32	0.0572	29.88	892.60

Table C-2. Timing and Speedups of the Level 1 Implementation (64 Wpns)

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
64	64	1	4.1830	1.00	3.96
64	64	2	0.6450	6.49	25.71
64	64	4	0.2143	19.52	77.37
64	64	8	0.0959	43.62	172.90
64	64	16	*0.0447	*93.58	*370.94
64	64	32	*0.0224	*186.74	*740.22
64	320	1	3.5310	1.00	37.88
64	320	2	1.6940	2.08	77.93
64	320	4	0.8408	4.20	157.01
64	320	8	0.4225	8.36	312.46
64	320	16	0.2139	16.51	617.19
64	320	32	0.1095	32.25	1205.62
64	640	1	6.7820	1.00	—
64	640	2	3.3800	2.01	—
64	640	4	1.6943	4.00	—
64	640	8	0.8515	7.96	—
64	640	16	0.4308	15.74	—
64	640	32	0.2200	30.83	—

Table C-3. Timing and Speedups of the Level 1 Implementation (128 Wpns)

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
128	128	1	14.5550	1.00	6.94
128	128	2	2.2250	6.54	45.40
128	128	4	0.8055	18.07	125.40
128	128	8	0.3605	40.37	280.20
128	128	16	0.1759	82.75	574.25
128	128	32	0.0871	167.11	1159.71
128	640	1	13.5720	1.00	—
128	640	2	6.7240	2.02	—
128	640	4	3.3478	4.05	—
128	640	8	1.6771	8.09	—
128	640	16	0.8424	16.11	—
128	640	32	0.4263	31.83	—
128	1280	1	26.8220	1.00	—
128	1280	2	13.3425	2.01	—
128	1280	4	6.6820	4.01	—
128	1280	8	3.3503	8.01	—
128	1280	16	1.6848	15.92	—
128	1280	32	0.8517	31.49	—

Table C-4. Timing and Speedups of the Level 2 Implementation (32 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
32	32	2	2	6	*0.036	*32.42	*65.33
32	32	2	4	10	*0.024	*53.13	*98.00
32	32	2	8	18	*0.030	*42.50	*78.40
32	32	4	2	12	*0.006	*212.50	*392.00
32	32	4	4	20	*0.014	*91.07	*168.00
32	32	8	2	24	*0.004	*318.75	*588.00
32	160	2	2	6	0.117	7.78	185.41
32	160	2	4	10	0.068	12.38	319.01
32	160	2	8	18	0.069	13.19	314.39
32	160	4	2	12	*0.009	*101.11	*2410.33
32	160	4	4	20	*0.035	*26.00	*619.80
32	160	8	2	24	*0.005	*182.00	*4338.60
32	320	2	2	6	0.205	8.34	249.06
32	320	2	4	10	0.121	14.12	421.96
32	320	2	8	18	0.093	18.38	549.00
32	320	4	2	12	*0.036	*47.47	*1418.25
32	320	4	4	20	0.062	27.56	823.50
32	320	8	2	24	*0.016	*106.81	*3191.06

Table C-5. Timing and Speedups of the Level 2 Implementation (64 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
64	64	2	2	6	0.244	17.14	67.95
64	64	2	4	10	0.160	26.14	103.63
64	64	2	8	18	0.129	32.43	128.53
64	64	4	2	12	0.029	144.24	571.76
64	64	4	4	20	0.060	69.72	276.35
64	64	8	2	24	0.010	418.38	1658.10
64	320	2	2	6	0.902	3.91	146.36
64	320	2	4	10	0.487	7.25	271.08
64	320	2	8	18	0.284	12.43	464.85
64	320	4	2	12	0.395	8.94	334.22
64	320	4	4	20	0.256	13.79	515.69
64	320	8	2	24	0.176	20.06	750.09
64	640	2	2	6	1.813	3.74	—
64	640	2	4	10	0.925	7.33	—
64	640	2	8	18	0.530	12.79	—
64	640	4	2	12	0.899	7.54	—
64	640	4	4	20	0.476	14.24	—
64	640	8	2	24	0.451	15.03	—

Table C-6. Timing and Speedups of the Level 2 Implementation (128 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
128	128	2	2	6	1.009	14.43	100.11
128	128	2	4	10	0.527	27.62	191.67
128	128	2	8	18	0.359	40.54	281.37
128	128	4	2	12	0.340	42.81	297.09
128	128	4	4	20	0.244	59.65	413.98
128	128	8	2	24	0.132	110.27	765.24
128	640	2	2	6	3.668	3.70	-
128	640	2	4	10	1.903	7.13	—
128	640	2	8	18	1.033	13.14	-
128	640	4	2	12	1.812	7.49	-
128	640	4	4	20	0.932	14.56	—
128	640	8	2	24	0.902	15.05	-
128	1280	2	2	6	8.378	3.20	-
128	1280	2	4	10	4.283	6.26	-
128	1280	2	8	18	2.237	11.99	—
128	1280	4	2	12	4.195	6.39	—
128	1280	4	4	20	2.145	12.50	-
128	1280	8	2	24	2.122	12.64	

Table C-7. Timing and Speedups of the Level 3 Implementation (32 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
32	32	2	2	6	0.1500	8.50	15.68
32	32	2	4	10	0.3360	3.79	7.00
32	32	2	8	18	0.9200	1.39	2.56
32	32	4	2	12	0.5158	2.47	4.56
32	32	4	4	20	0.6073	2.10	1.12
32	32	8	2	24	0.3403	3.75	6.91
32	160	2	2	6	0.2955	3.08	73.41
32	160	2	4	10	0.5440	1.67	39.88
32	160	2	8	18	0.8465	1.08	25.63
32	160	4	2	12	0.5275	1.73	41.12
32	160	4	4	20	0.6650	1.37	32.65
32	160	8	2	24	0.3976	2.29	54.56
32	320	2	2	6	0.4940	3.46	103.35
32	320	2	4	10	0.7965	2.15	64.10
32	320	2	8	18	1.0390	1.64	49.14
32	320	4	2	12	0.6968	2.45	73.27
32	320	4	4	20	0.8420	2.03	60.64
32	320	8	2	24	0.5058	3.38	100.94

Table C-8. Timing and Speedups of the Level 3 Implementation (64 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
64	64	2	2	6	0.4485	9.33	36.97
64	64	2	4	10	0.9235	4.53	17.95
64	64	2	8	18	1.7630	2.37	9.40
64	64	4	2	12	1.0165	4.12	16.31
64	64	4	4	20	1.1958	3.50	13.87
64	64	8	2	24	0.6671	6.27	24.86
64	320	2	2	6	1.1194	3.15	117.93
64	320	2	4	10	1.9175	1.84	68.85
64	320	2	8	18	2.6600	1.33	49.63
64	320	4	2	12	1.7131	2.06	77.06
64	320	4	4	20	2.0381	1.73	64.77
64	320	8	2	24	1.1914	2.96	110.81
64	640	2	2	6	2.0045	3.38	—
64	640	2	4	10	3.2400	2.09	—
64	640	2	8	18	4.1520	1.63	—
64	640	4	2	12	2.8298	2.40	—
64	640	4	4	20	3.4108	1.99	—
64	640	8	2	24	2.0333	3.34	—

Table C-9. Timing and Speedups of the Level 3 Implementation (128 Wpns

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Time (sec)	$S_{B\&L}$	S_{Sort}
128	128	2	2	6	1.5355	9.48	65.78
128	128	2	4	10	3.0785	4.73	32.81
128	128	2	8	18	5.6275	2.59	17.95
128	128	4	2	12	3.3035	4.41	30.58
128	128	4	4	20	3.8353	3.80	26.34
128	128	8	2	24	2.1064	6.91	47.95
128	640	2	2	6	4.5330	2.99	—
128	640	2	4	10	7.8575	1.73	
128	640	2	8	18	11.3755	1.19	
128	640	4	2	12	7.2560	1.87	
128	640	4	4	20	8.5495	1.59	
128	640	8	2	24	4.9200	2.76	
128	1280	2	2	6	8.7008	3.08	—
128	1280	2	4	10	14.9796	1.79	
128	1280	2	8	18	20.3846	1.32	
128	1280	4	2	12	13.4023	2.00	—
128	1280	4	4	20	15.3583	1.75	
128	1280	8	2	24	8.9406	3.00	—

Table C-10. Timing and Speedups of the Level 4 Implementation (32 Wpns

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
32	32	2	1.641	0.78	1.43
32	32	4	1.713	0.74	1.37
32	32	8	1.934	0.66	1.22
32	32	16	3.918	0.33	0.60
32	32	32	12.798	0.10	0.18
32	160	2	1.513	0.60	14.34
32	160	4	1.285	0.71	16.88
32	160	8	2.176	0.42	9.97
32	160	16	7.354	0.12	2.95
32	160	32	21.327	0.04	1.02
32	320	2	2.340	0.73	21.82
32	320	4	1.461	1.17	34.95
32	320	8	2.091	0.82	24.42
32	320	16	7.654	0.28	6.67

Table C-11. Timing and Speedups of the Level 4 Implementation (64 Wpns)

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
64	64	2	2.594	1.61	6.39
64	64	4	2.540	1.65	6.53
64	64	8	3.296	1.27	5.03
64	64	16	4.589	0.91	3.61
64	64	32	12.463	0.34	1.33
64	320	2	5.884	0.60	22.44
64	320	4	3.427	1.03	38.52
64	320	8	4.022	0.88	32.82
64	320	16	9.100	0.39	14.51
64	320	32	26.155	0.14	5.05
64	640	2	9.585	1.41	—
64	640	4	6.020	1.13	—
64	640	8	6.106	1.11	—
64	640	16	11.203	0.61	—

Table C-12. Timing and Speedups of the Level 4 Implementation (128 Wpns)

Weap	Targ	Processors	Time (sec)	$S_{B\&L}$	S_{Sort}
128	128	2	17.808	0.82	5.67
128	128	4	15.433	0.94	6.55
128	128	8	16.619	0.88	6.08
128	128	16	13.246	1.10	7.63
128	128	32	24.843	0.59	4.07
128	640	2	20.321	0.67	—
128	640	4	11.739	1.16	—
128	640	8	11.865	1.14	—
128	640	16	16.579	0.82	—
128	1280	2	33.239	0.81	—
128	1280	4	22.152	1.21	—
128	1280	8	18.657	1.44	—
128	1280	16	26.61	1.01	—

Table C-13. Assignment Results of the Level 1 Implementation (32 Wpns)

Weap	Targ	Processors	Cost	% Effective	% Wasted
32	32	1	2660.8	100.0	0.0
32	32	2	1766.4	74.4	26.5
32	32	4	1550.4	67.5	32.5
32	32	8	1528.0	65.6	34.4
32	32	16	1476.8	64.4	35.6
32	32	32	1473.6	63.1	36.9
32	160	1	384.0	100.0	0.0
32	160	2	380.8	93.8	6.2
32	160	4	376.0	92.5	7.5
32	160	8	376.0	91.3	8.7
32	160	16	376.0	90.6	9.4
32	160	32	376.0	90.6	9.4
32	320	1	278.4	100.0	0.0
32	320	2	276.8	95.0	5.0
32	320	4	276.8	93.8	6.2
32	320	8	276.8	93.8	6.2
32	320	16	276.8	93.8	6.2
32	320	32	276.8	93.8	6.2

Table C-14. Assignment Results of the Level 1 Implementation (64 Wpns)

Weap	Targ	Processors	Cost	% Effective	% Wasted
64	64	1	2016.0	100.0	0.0
64	64	2	1214.4	75.6	24.4
64	64	4	1337.6	71.3	28.7
64	64	8	1310.4	69.7	30.3
64	64	16	1296.0	68.1	31.9
64	64	32	1289.6	67.5	32.5
64	320	1	550.4	100.0	0.0
64	320	2	547.2	93.4	6.6
64	320	4	545.6	90.0	10.0
64	320	8	545.6	89.1	10.9
64	320	16	545.6	88.8	11.2
64	320	32	545.6	88.8	11.2
64	640	1	486.4	100.0	0.0
64	640	2	486.4	92.8	7.2
64	640	4	486.4	91.3	8.7
64	640	8	486.4	89.7	10.3
64	640	16	486.4	88.4	11.6
64	640	32	486.4	87.8	12.2

Table C-15. Assignment Results of the Level 1 Implementation (128 Wpns)

Weap	Targ	Processors	Cost	% Effective	% Wasted
128	128	1	2158.4	100.0	0.0
128	128	2	1668.8	74.4	25.6
128	128	4	1601.6	68.1	31.9
128	128	8	1566.4	65.2	34.8
128	128	16	1558.4	64.1	35.9
128	128	32	1552.0	63.8	36.2
128	640	1	992.0	100.0	0.0
128	640	2	990.4	90.6	9.4
128	640	4	990.4	85.0	15.0
128	640	8	990.4	82.7	17.3
128	640	16	990.4	81.7	18.3
128	640	32	990.4	81.1	18.9
128	1280	1	937.6	100.0	0.0
128	1280	2	936.0	89.7	10.3
128	1280	4	936.0	85.3	14.7
128	1280	8	936.0	82.2	17.8
128	1280	16	936.0	80.9	19.1
128	1280	32	936.0	80.5	19.4

Table C-16. Assignment Results of the Level 2 Implementation (32 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Idle	% Wasted
32	32	2	2	6	1113.6	68.1	16.3	15.6
32	32	2	4	10	1008.0	65.6	20.0	14.4
32	32	2	8	18	915.2	64.4	21.9	13.7
32	32	4	2	12	1406.4	65.6	5.0	29.4
32	32	4	4	20	1296.0	64.4	6.9	28.7
32	32	8	2	24	1412.8	64.4	2.5	33.1
32	160	2	2	6	368.0	92.5	1.9	5.6
32	160	2	4	10	361.6	91.2	3.8	5.0
32	160	2	8	18	355.2	90.6	5.0	4.4
32	160	4	2	12	369.6	91.2	1.9	6.9
32	160	4	4	20	363.2	90.6	3.1	6.3
32	160	8	2	24	369.6	90.6	1.3	8.1
32	320	2	2	6	272.0	93.8	1.9	4.4
32	320	2	4	10	272.0	93.8	1.9	4.4
32	320	2	8	18	272.0	93.8	1.9	4.4
32	320	4	2	12	276.8	93.8	0.0	6.3
32	320	4	4	20	276.8	93.8	0.0	6.3
32	320	8	2	24	276.8	93.8	0.0	6.3

Table C-17. Assignment Results of the Level 2 Implementation (64 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Idle	% Wasted
64	64	2	2	6	1158.4	71.3	10.4	17.9
64	64	2	4	10	1099.2	69.7	12.8	17.5
64	64	2	8	18	1033.6	68.1	15.6	16.3
64	64	4	2	12	1236.8	69.7	3.8	26.6
64	64	4	4	20	1171.2	68.1	6.6	25.3
64	64	8	2	24	1220.8	68.1	2.8	29.1
64	320	2	2	6	523.2	90.0	3.8	6.3
64	320	2	4	10	520.0	89.1	4.7	6.3
64	320	2	8	18	518.4	88.8	5.0	6.3
64	320	4	2	12	539.2	89.1	1.3	9.7
64	320	4	4	20	537.6	88.8	1.6	9.7
64	320	8	2	24	544.0	88.8	0.3	10.9
64	640	2	2	6	476.8	91.3	1.9	6.8
64	640	2	4	10	468.8	89.7	3.4	6.8
64	640	2	8	18	460.8	88.1	5.0	6.8
64	640	4	2	12	478.4	89.7	1.6	8.8
64	640	4	4	20	470.4	88.1	3.1	8.8
64	640	8	2	24	478.4	88.1	1.6	10.3

Table C-18. Assignment Results of the Level 2 Implementation (128 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Idle	% Wasted
128	128	2	2	6	1361.6	90.8	15.4	27.1
128	128	2	4	10	1259.2	65.2	15.8	19.1
128	128	2	8	18	1203.2	64.1	17.8	18.1
128	128	4	2	12	1446.4	65.2	6.3	28.6
128	128	4	4	20	1372.8	64.1	8.9	27.0
128	128	8	2	24	1483.2	64.1	3.1	32.8
128	640	2	2	6	928.0	85.0	6.1	8.9
128	640	2	4	10	902.4	82.7	8.6	8.8
128	640	2	8	18	889.6	81.7	9.8	8.4
128	640	4	2	12	963.2	82.7	2.7	14.7
128	640	4	4	20	948.8	81.7	4.1	14.2
128	640	8	2	24	976.0	81.7	1.4	16.9
128	1280	2	2	6	881.6	85.3	5.3	9.4
128	1280	2	4	10	846.4	82.2	8.8	9.1
128	1280	2	8	18	833.6	80.9	10.0	9.1
128	1280	4	2	12	900.8	82.2	3.4	14.4
128	1280	4	4	20	884.8	80.9	5.0	14.1
128	1280	8	2	24	920.0	80.9	1.6	17.5

Table C-19. Assignment Results of the Level 3 Implementation (32 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Wasted
32	32	2	2	6	9596.8	76.3	23.7
32	32	2	4	10	9857.6	78.1	21.9
32	32	2	8	18	8180.8	72.5	27.5
32	32	4	2	12	4302.4	66.9	33.1
32	32	4	4	20	4385.6	66.9	33.1
32	32	8	2	24	2510.4	63.8	36.2
32	160	2	2	6	672.0	93.8	6.2
32	160	2	4	10	1276.8	93.8	6.2
32	160	2	8	18	1916.8	93.8	6.2
32	160	4	2	12	1104.0	91.9	8.1
32	160	4	4	20	1656.0	92.5	7.5
32	160	8	2	24	964.8	91.3	8.7
32	320	2	2	6	1332.8	95.0	5.0
32	320	2	4	10	1332.8	95.0	5.0
32	320	2	8	18	1332.8	95.0	5.0
32	320	4	2	12	276.8	93.8	6.2
32	320	4	4	20	276.8	93.8	6.2
32	320	8	2	24	276.8	93.8	6.2

Table C-20. Assignment Results of the Level 3 Implementation (64 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Wasted
64	64	2	2	6	9766.4	76.9	23.1
64	64	2	4	10	10984.0	75.0	25.0
64	64	2	8	18	10337.6	73.8	26.2
64	64	4	2	12	4660.8	70.3	29.7
64	64	4	4	20	7102.4	69.1	30.9
64	64	8	2	24	4192.0	68.4	31.6
64	320	2	2	6	4758.0	93.4	6.6
64	320	2	4	10	5696.0	92.2	7.8
64	320	2	8	18	5928.0	92.2	7.8
64	320	4	2	12	1764.0	89.8	10.2
64	320	4	4	20	1972.0	89.8	10.2
64	320	8	2	24	1222.0	89.1	10.9
64	640	2	2	6	1780.8	92.5	7.5
64	640	2	4	10	4190.4	91.9	8.1
64	640	2	8	18	5326.4	90.6	9.4
64	640	4	2	12	2206.4	90.6	9.4
64	640	4	4	20	2881.6	89.7	10.3
64	640	8	2	24	1960.0	89.1	10.9

Table C-21. Assignment Results of the Level 3 Implementation (128 Wpns)

Weap	Targ	Cntrl	Proc/Cntrl	Tot Proc	Cost	% Effective	% Wasted
128	128	2	2	6	16811.2	72.3	27.7
128	128	2	4	10	20294.0	69.5	30.5
128	128	2	8	18	17078.4	69.4	30.6
128	128	4	2	12	9393.6	66.4	33.6
128	128	4	4	20	13936.0	65.8	34.2
128	128	8	2	24	9993.3	65.9	34.1
128	640	2	2	6	12964.8	87.8	12.2
128	640	2	4	10	13024.0	87.5	12.5
128	640	2	8	18	14980.8	86.4	13.6
128	640	4	2	12	6249.6	83.6	16.4
128	640	4	4	20	8673.6	82.8	17.2
128	640	8	2	24	3081.6	81.9	18.1
128	1280	2	2	6	11006.7	87.0	13.0
128	1280	2	4	10	14636.0	86.1	13.9
128	1280	2	8	18	12617.3	84.9	15.1
128	1280	4	2	12	7730.7	82.7	17.3
128	1280	4	4	20	8046.7	81.8	18.2
128	1280	8	2	24	2969.3	81.3	18.7

Table C-22. Assignment Results of the Level 4 Implementation (32 Wpns)

Weap	Targ	Processors	Cost	% Effective
32	32	2	2660.8	100.0
32	32	4	2660.8	100.0
32	32	8	2660.8	100.0
32	32	16	2660.8	100.0
32	32	32	2660.8	100.0
32	160	2	384.0	100.0
32	160	4	384.0	100.0
32	160	8	384.0	100.0
32	160	16	384.0	100.0
32	160	32	384.0	100.0
32	320	2	278.4	100.0
32	320	4	278.4	100.0
32	320	8	278.4	100.0
32	320	16	278.4	100.0

Table C-23. Assignment Results of the Level 4 Implementation (64 Wpns)

Weap	Targ	Processors	Cost	% Effective
64	64	2	2016.0	100.0
64	64	4	2016.0	100.0
64	64	8	2016.0	100.0
64	64	16	2016.0	100.0
64	64	32	2016.0	100.0
64	320	2	550.4	100.0
64	320	4	550.4	100.0
64	320	8	550.4	100.0
64	320	16	550.4	100.0
64	320	32	550.4	100.0
64	640	2	486.4	100.0
64	640	4	486.4	100.0
64	640	8	486.4	100.0
64	640	16	486.4	100.0

Table C-24. Assignment Results of the Level 4 Implementation (128 Wpns)

Weap	Targ	Processors	Cost	% Effective
128	128	2	2158.4	100.0
128	128	4	2158.4	100.0
128	128	8	2158.4	100.0
128	128	16	2158.4	100.0
128	128	32	2158.4	100.0
128	640	2	992.0	100.0
128	640	4	992.0	100.0
128	640	8	992.0	100.0
128	640	16	992.0	100.0
128	1280	2	937.6	100.0
128	1280	4	937.6	100.0
128	1280	8	937.6	100.0
128	1280	16	937.6	100.0

Bibliography

AdF85. Adam, J.A. and M.A. Fischetti. "Star Wars, SDI: The Grand Experiment," *IEEE Spectrum*, 22: pp. 34-35 (September 1985).

AdW85. Adam, J.A. and P. Wallich. "Mind Boggling Complexity," *IEEE Spectrum*, 22: pp. 37-46 (September 1985).

BaG77. Barr, R.S., F. Glover, and D. Klingman. "The Alternating Basis Algorithm for Assignment Problems," *Mathematical Programming*, 13: pp. 1-13 (1977).

Ber81. Bertsekas, D.P. "A New Algorithm for the Assignment Problem," *Mathematical Programming*, 21: pp. 152-171 (1981).

BoL71a. Bourgeois, L. and J. Lassalle. "An Extension of the Munkres Algorithm for the Assignment Problem to Rectangular Matrices," *Communications of the ACM*, 14: pp. 802-804 (December 1971).

BoL71b. ——. "Algorithm 415: Algorithm for the Assignment Problem (Rectangular Matrices)[H]" *Communications of the ACM*, 14: pp. 805-806 (December 1971).

BoW85. Bosma, J.T. and R.C. Wheelan *Guide to the Strategic Defense Initiative*. Arlington, Virginia: Pasha Publications, 1985.

CaT80. Carpaneto, G. and P. Toth. "Algorithm 548 Solution of the Assignment Problem [H]," *ACM Transactions on Mathematical Software*, 6: pp. 104-111 (March 1980).

Chu57. Churchman, C.W., R.L. Ackoff, and E.L. Arnoff. *Introduction to Operations Research*, New York: John Wiley and Sons Incorporated, 1957.

CoS87. Cody, R. and J. Smith. *Applied Statistics and the SAS Programming Language*. New York: Elsevier Science Publishing Company, 1987.

Cur87. Curkendall, D. In conversation with N. J. Davis. Jet Propulsion Laboratory, Pasadena CA, 12 August 1987.

Cve87. Cvetanovic, Z. "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," *IEEE Transactions on Computers*, C-36: pp. 421-432 (April 1987).

Dan63. Dantzig, G.B. *Linear Programming and Extensions*. Princeton, New Jersey: Princeton University Press, 1963.

Den86. Denning, P.J. "Parallel Computing and its Evolution," *Communications of the ACM*, 29: pp. 1163-1167 (December 1986).

DrF85. Drell, D., P.J. Forley, and D. Holloway. *The Reagan Strategic Defense Initiative, A Technical, Political and Arms Control Assessment*. Cambridge, Massachusetts: Ballenger Publishing, 1985.

EbB86. Eberhardt, D. S. and D. Baganoff. *Multiple Grid Problems on Concurrent Processing Computers.* NASA Technical Memorandum 86675. NASA Ames Research Center, February 1986 (N86-32112).

FeK85. Felten, E., S. Karlin, and S. Otto. "The Traveling Salesman Problem on a Hypercubic, MIMD Computer," *Proceedings of the 1985 International Conference on Parallel Processing.* pp. 6-10. New York: IEEE Computer Society Press, 1985.

Fel85. Feldman, M. *Data Structures with Ada.* Reston, Virginia: Reston Publishing Company, 1985.

Fen81. Feng, T. "A Survey of Interconnection Networks," *Computer,* pp. 12-27 (December 1981).

Fly66. Flynn, M.J. "Very High Speed Computing," *Proceedings of the IEEE,* 54: pp. 1901-1909 (December 1966).

Fox84. Fox, G.C. "Concurrent Processing for Scientific Calculations," *Proceedings of the COMPCON 1984 IEEE Conference.* pp. 70-73 IEEE Press, New York, 1984.

FoO84. Fox, G.C. and S.W. Otto. "Algorithms for Concurrent Processors," *Physics Today.* pp. 13-20 (May 1984).

Fre86. Frenkel, K.A. "Evaluating Two Massively Parallel Machines," *Communications of the ACM,* 29: pp. 752-758 (August 1986).

GlK74. Glover, F., D. Karney, and D. Klingman. "Implementation and Computational Comparisons of Primal, Dual, and Primal-Dual Computer Codes for Minimum Cost Network Flow Problems," *Networks,* 4: pp. 191-212 (1974).

Got87. Gottschalk, T. "Boost-Phase Track Initiation Algorithms Implemented on Hypercube," Presented at the *Workshop on Advanced Computing for Strategic Defense,* Arlington, VA, 23 March 1987.

HaL82. Haynes, L.S., R.L. Lau, D.P. Siewiorek, and D.W. Mizell. "A Survey of Highly Parallel Computing," *IEEE Computer,* 15: pp. 9-26 (January 1982).

HaM86. Hayes, J.P., T.N. Mudge, Q.F. Stout, S. Colley, and J. Palmer. "Architecture of a Hypercube Supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing.* pp. 653-665. Washington D.C.:IEEE Computer Society Press, 1986.

Hat75. Hatch, R.S. "Bench Marks Comparing Transportation Codes Based on Primal Simplex and Primal-Dual Algorithms," *Operations Research,* 23: pp. 1167-1171 (1975).

Hil87. Hillis, W.D. "The Connection Machine" *Scientific American,* 256: pp. 108-115 (June 1987).

HoZ83. Horowitz, E. and A. Zorat. "Divide and Conquer for Parallel Processing," *IEEE Transactions on Computers*, C-32: pp. 582-585 (June 1983).

Hun83. Hung, M.S. "A Polynomial Simplex Method for the Assignment Problem," *Operations Research 31*: pp. 595-600 (May-June 1983).

HwB84. Hwang, K. and F.A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill 1984.

Ign82. Ignizio, J.P. *Linear Programming in Single and Multiple-Objective Systems*. Englewood Cliffs, New Jersey: Prentice-Hall Incorporated, 1982.

Int86. *iPSC System Overview Manual*. Order Number 310610-001. Intel Scientific Computers, Beaverton, Oregon, November 1986.

KeR78. Kernigham, B. and D. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall Incorporated, 1978.

Kre68. Krekó, B. *Linear Programming*. New York: American Elsevier Publishing, 1968.

Kuh55. Kuhn, H.W. "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, 2: pp. 83-97 (1955).

Kur62. Kurtzberg, J. M. "On Approximation Methods for the Assignment Problem," *Journal of the ACM*, 9: pp. 419-439 (1962).

LaM69. Lawler, E.L. and J.M. Moore. "A Functional Equation and its Application to Resource Allocation and Sequencing Problems," *Management Science*, 16: pp. 77-84 (September 1969).

Lin85. L., Herbert. "The Development of Software for Ballistic Missile Defense," *Scientific American*, 253: pp. 46-53 (December 1985).

MaN86. Mazzola, J.B. and A.W. Neebe. "Resource Constrained Assignment Scheduling," *Operations Research*, 34: pp. 560-572 (July - August 1986).

McG83. McGinnis, L.F. "Implementation and Testing of a Primal-Dual Algorithm for the Assignment Problem," *Operations Research*, 31: pp. 277-291 (1983).

McS85. McMillen, R.J. and H.J. Siegel. "Evaluation of Cube and Data Manipulator Networks," *Journal of Parallel and Distributed Computing*. pp. 79-107 (February 1985).

Mra86. Mraz, R.T. *Performance Evaluation of Parallel Branch and Bound Search with the Intel iPSC Hypercube Computer*. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, Ohio, December, 1986.

MuA87. Mudge, T.N. and T.S. Abdel-Rahman. "Vision Algorithms for Hypercube Machines," *Journal of Parallel and Distributed Computing*, 4: pp. 79-94 (February 1987).

Mun57. Munkres, J. "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, 5: pp. 32-38 (March 1957).

Noz86. Nozette, S. "The Commercial Potential of SDI," *Promise or Peril, The Strategic Defense Initiative*, edited by Z. Brzezinski. Washington, DC: Ethics and Public Policy Center, 1986.

Odo85. Odom, P. *A Method for Improving Technology Research and Development Decisions Regarding BMD and ASAT, Volume III - User Manual and Simulation Description: Final Report*, Contract DAAH01-84-C-0486. Huntsville, Alabama: DESE Research and Engineering, Incorporated, July 1985 (AD-B093915).

Par85. Parnas, D.L. "Software Aspects of Strategic Defense Systems," *Communications of the ACM*, 28: pp. 1326-27 (December 1985).

Pea77. Pease, M.C. "The Indirect Binary N-cube Microprocessor Array," *IEEE Transactions on Computers*, C-25: pp. 458-473 (May 1977).

Qui87. Quinn, M.J. *Designing Efficient Algorithms for Parallel Computers* New York: McGraw-Hill 1987.

Rea83. Reagan, R.R. "Defense Spending and Defense Technology," *Weekly Compilation of Presidential Documents*, 19: pp. 423-466 (28 March 1983).

RiS84. Riganati, J.P. and P.B. Schneck. "Supercomputing," *Computer*, pp. 97-113 (October 1984).

Saa86. Saad, Y. *Gaussian Elimination on Hypercubes*. YALEU/DCS/RR0462. New Haven, CN: Yale University, Department of Computer Science, March 1986 (AD-A169293).

SaS85. Saad, Y. and M.H. Schultz. *Topological Properties of Hypercubes*. YALEU/DCS/RR-389. New Haven, CN: Yale University, Department of Computer Science, June 1985 (AD-A156777).

SaN85. Saltz, J.H. and V.K. Naik. *Towards Developing Robust Algorithms for Solving Differential Equations on MIMD Machines*. NASA Report N86-11856, September 1985.

San87. Sandell, N. "Weapon-Target Assignment Algorithms for SDI." Presented at the *Workshop on Advanced Computing for Strategic Defense*, Arlington, VA, 23 March 1987.

SeB82. Selim, A.G., D.T. Barnard, and R.J. Doran. "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4: pp. 192-202 (March 1982).

Sei85. Seitz, C.L. "The Cosmic Cube," *Communication of the ACM*, 28: pp. 22-23 (January 1985).

SeD85. Seward, W.D., and N.J. Davis IV. "Opportunities and Issues for Parallel Processing in SDI Battle Management/C3," Presented at the AIAA Computers in Aerospace V Conference, October 1985.

SiS84. Seigel, H.J., T. Schwederski, N.J. Davis IV, and J. T. Kuehn. "PASM: A Reconfigurable Parallel System for Image Processing," *Computer Architecture News*, pp. 7-19 (September 1984).

Smi82. Smith, D.K. *Network Optimisation Practice: A Computational Guide*. New York: John Wiley and Sons, 1982.

SrT72. Srinivasan, V. and G.L. Thompson. "Accelerated Algorithms for Labeling and Relabeling of Trees with Applications to Distribution Problems," *Journal of the ACM*, 19: pp. 712-726 (1972).

SrT73. Srinivasan, V. and G.L. Thompson "Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Problem," *Journal of the ACM*, 20: pp. 194-213 (1973).

WaL85. Wah, B.W., G. Li, and C.F. Yu. "Multiprocessing of Combinatorial Search Problems," *Computer*, 18: pp. 93 - 108. (June 1985).

Vita

Captain Barry A. Carpenter was born on 19 October 1954 at Fort Lee, Virginia. After graduating from Greer High School, Greer, South Carolina in 1972, he joined the Air Force. He held various positions as a flight simulator technician at Pease AFB, New Hampshire, and Ramstein AB, West Germany. In 1983, he graduated Summa Cum Laude from North Carolina State University, Raleigh, North Carolina with a Bachelor of Science in Electrical Engineering. He gained his commission through Officer Training School (OTS), after which he was assigned to the Shuttle Test Group at Vandenberg AFB, California. His primary duties at Vandenberg included design, development and testing of ground electrical, photographic lighting, and high-speed camera systems for the space shuttle launch pad. In May of 1986, Captain Carpenter entered the Computer Engineering program at the School of Engineering, Air Force Institute of Technology.

Permanent address: 104 McDaniel Ave.
Greer, South Carolina
29651

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCE/ENG/87D-2	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-6583	7b ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable): OSD/SDIO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code): Pentagon, Washington D.C. 20301-7100	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
IMPLEMENTATION AND PERFORMANCE ANALYSIS OF PARALLEL ASSIGNMENT ALGORITHMS ON A HYPERCUBE COMPUTER

12 PERSONAL AUTHOR(S)
Barry A. Carpenter, B.S., Capt, USAF

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
M.S. thesis	FROM _____ TO _____	1987 December	164

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	performance tests, computer communications, parallel processing, algorithms, antimissile defense
12	05		

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: Nathaniel J. Davis IV, Capt, USA
Assistant Professor of Electrical Engineering

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Nathaniel J. Davis, Capt, USA	22b TELEPHONE (Include Area Code): (513) 255-2024; 22c OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete* SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

ABSTRACT

The process of effectively coordinating and controlling resources during a military engagement is known as Battle Management/ Command, Control, and Communications (BM/C3). One key task of BM/C3 is allocating weapons to destroy targets. The focus of this research is on developing parallel methods to achieve fast and cost effective assignment of weapons to targets. Using the sequential Hungarian method for solving the assignment problem as a basis, this report presents the development of four parallel assignment algorithms implemented on the Intel iPSC hypercube computer.

The first approach partitions the problem space into smaller, independent sub-problems and assigns each to a processing node the hypercube. The second and third approaches also partition the oblem space but they assign each partition to a group of processing des. Each group is controlled by a separate node which further subdi ides the partition among members of the group. In the second approa h, the control node acts as an arbitrator to eliminate the redundant assignment of weapons by selecting the least costly weapon allocation and idling the more costly redundant allocations. The third approach eliminates redundant weapon allocations by also selecting the least costly weapon allocations, but directs additional processing to reallocate the more costly weapons. The fourth approach is a parallel implementation of the Hungarian method, where certain subtasks of the algorithm are performed in parallel. This approach produces an optimal assignment instead of the sub-optimal assignment generally obtained using either of the three heuristic methods.

The relative performance of the four approaches is compared by varying the number of weapons and targets, the number of processors, and the size of the problem partitions. The first and second approaches produce significantly faster assignment solutions than those possible with the baseline sequential methods. The third and fourth approaches yield slower solutions, but are still faster than sequential methods of assignment.

END
DATE
FILMED
4-88
DTIC